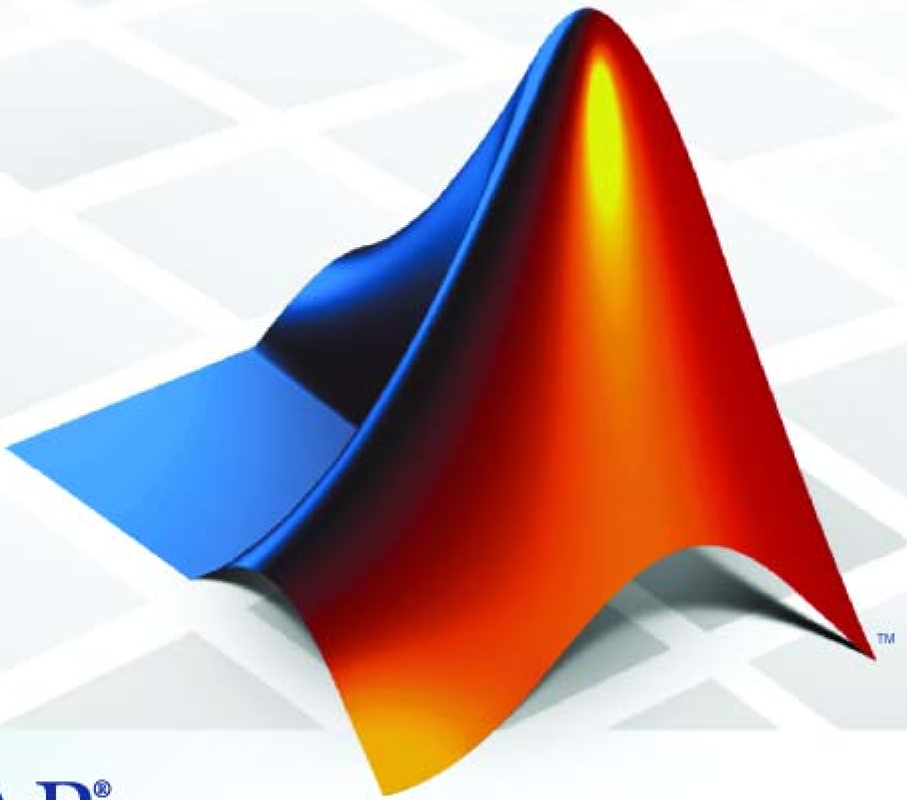
Simulink® 7
Writing S-Functions



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Writing S-Functions

Revision History

October 1998	First printing	Revised for Version 3.0 (Release R11)
November 2000	Second printing	Revised for Version 4.0 (Release R12)
July 2002	Third printing	Revised for Version 5.0 Release R13)
April 2003	Online only	Revised for Version 5.1 (Release R13SP1)
April 2004	Online only	Revised for Version 5.1.1 (Release R13SP1+)
June 2004	Online only	Revised for Version 6.0 (Release R14)
October 2004	Online only	Revised for Version 6.1 (Release R14SP1)
March 2005	Online only	Revised for Version 6.2 (Release R14SP2)
September 2005	Online Only	Revised for Version 6.3 (Release R14SP3)
March 2006	Online only	Revised for Version 6.4 (Release 2006a)
September 2006	Online only	Revised for Version 6.5 (Release 2006b)
March 2007	Online only	Revised for Version 6.6 (Release 2007a)
September 2007	Online only	Revised for Version 7.0 (Release 2007b)
March 2008	Online only	Revised for Version 7.1 (Release 2008a)
October 2008	Online only	Revised for Version 7.2 (Release 2008b)
March 2009	Online only	Revised for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.4 (Release 2009b)

Writing S-Functions

1 Overview of S-Functions

5 Creating C++ S-Functions

6 Creating Fortran S-Functions

Writing S-Functions

Overview of S-Functions

What Is an S-Function?

S-functions (system-functions) provide a powerful mechanism for extending the capabilities of the Simulink® environment. An *S-function* is a computer language description of a Simulink block written in MATLAB, C, C++, or Fortran. C, C++, and Fortran S-functions are compiled as MEX-files using the `mex` utility (see “Building MEX-Files” in *MATLAB External Interfaces*). As with other MEX-files, S-functions are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute.

S-functions use a special calling syntax called the S-function API that enables you to interact with the Simulink engine. This interaction is very similar to the interaction that takes place between the engine and built-in Simulink blocks.

S-functions follow a general form and can accommodate continuous, discrete, and hybrid systems. By following a set of simple rules, you can implement an algorithm in an S-function and use the S-Function block to add it to a Simulink model. After you write your S-function and place its name in an S-Function block (available in the User-Defined Functions block library), you can customize the user interface using masking (see “Creating Custom Blocks”).

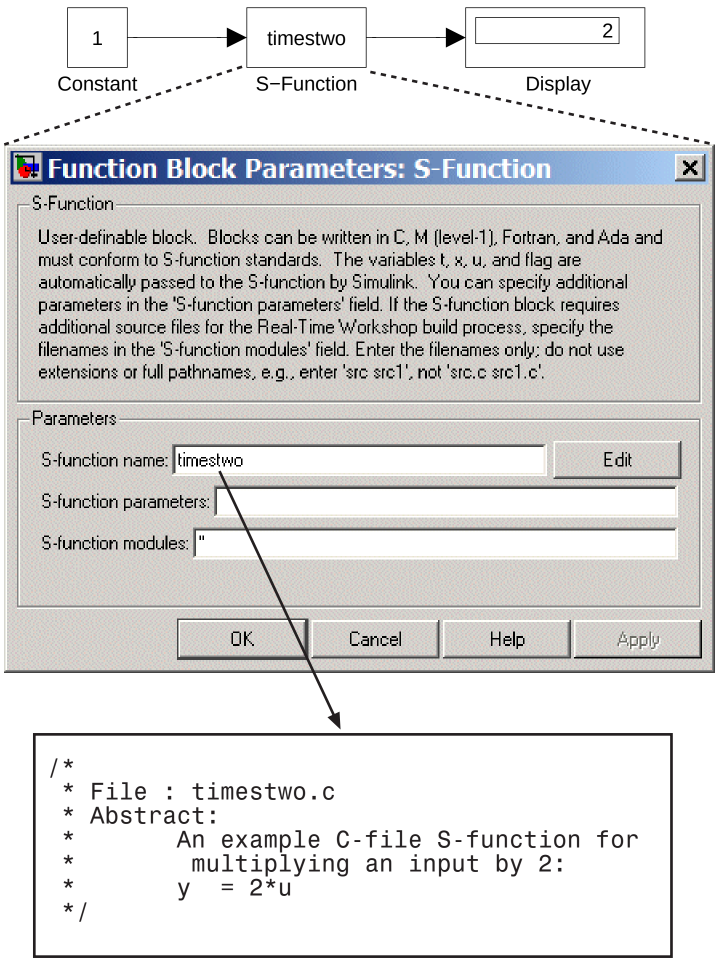
You can use S-functions with the Real-Time Workshop® product. You can also customize the code generated for S-functions by writing a Target Language Compiler (TLC) file. See “Integrating External Code With Generated C and C++ Code” in the *Real-Time Workshop User’s Guide* for more information.

Using S-Functions in Models

In this section...

Overview

To incorporate a C MEX S-function or legacy Level-1 M-file S-function into a Simulink model, drag an S-Function block from the User-Defined Functions block library into the model. Then specify the name of the S-function in the **S-function name** field of the S-Function block's Block Parameters dialog box, as illustrated in the following figure.



User-definable block. Blocks can be written in C, M (level-1), Fortran, and Ada and must conform to S-function standards. The variables t, x, u, and flag are automatically passed to the S-function by Simulink. You can specify additional parameters in the 'S-function parameters' field. If the S-function block requires additional source files for the Real-Time Workshop build process, specify the filenames in the 'S-function modules' field. Enter the filenames only; do not use extensions or full pathnames, e.g., enter 'src src1', not 'src.c src1.c'.

S-function name: timestwo

S-function modules: ''

```
/*
 * File : timestwo.c
 * Abstract:
 *       An example C-file S-function for
 *        multiplying an input by 2:
 *       y  = 2*u
 */
```

In this example, the model contains an S-Function block that references an instance of the C MEX-file for the S-function `timestwo.c`.

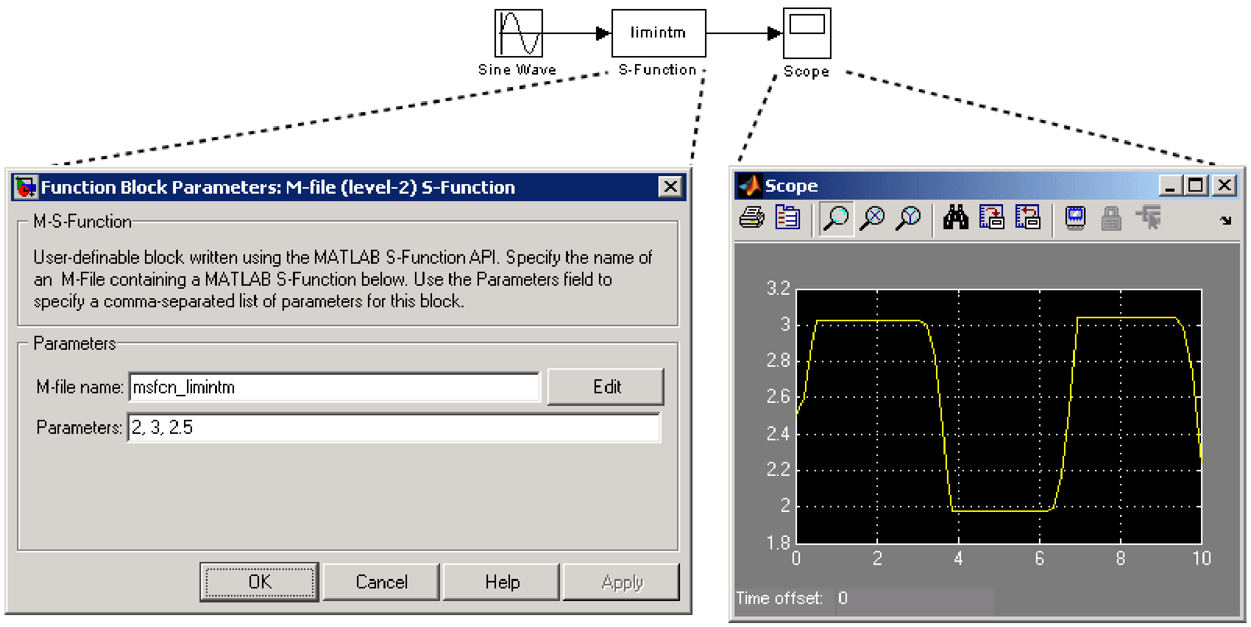
Note If the MATLAB path includes a C MEX-file and an M-file having the same name referenced by an S-Function block, the S-Function block uses the C MEX-file.

To incorporate a Level-2 M-file S-function into a model, drag a Level-2 M-file S-Function block from the User-Defined Functions library into the model. Specify the name of the S-function in the **M-file name** field.

Passing Parameters to S-Functions

The S-Function block's **S-function parameters** and Level-2 M-file S-Function block's **Parameters** fields allow you to specify parameter values to pass to the corresponding S-function. To use these fields, 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 MATLAB or model workspace, or MATLAB expressions.

The following example illustrates usage of the **Parameters** field to enter user-defined parameters for a Level-2 M-file S-function.



Function Block Parameters: M-file (level-2) S-Function

M-S-Function

User-definable block written using the MATLAB S-Function API. Specify the name of an M-File containing a MATLAB S-Function below. Use the Parameters field to specify a comma-separated list of parameters for this block.

Parameters

M-file name: msfcn_limintm

Parameters: 2, 3, 2.5

The model in this example incorporates the sample S-function

```
matlabroot/toolbox/simulink/blocks/msfcn_limintm.m
```

The `msfcn_limintm.m` S-function accepts three parameters: a lower bound, an upper bound, and an initial condition. The S-function 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 3-18 and "Error Handling" on page 8-69 for information on how to access user-specified parameters in an S-function.

You can use the 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 a discussion on masking, see "Working with Block Masks" in *Using Simulink*.

When to Use an S-Function

You can use S-functions for a variety of applications, including:

- Creating new general purpose blocks
- Adding blocks that represent hardware device drivers
- Incorporating existing C code into a simulation (see "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" on page 4-55)
- Describing a system as a set of mathematical equations
- Using graphical animations (see the inverted pendulum demo, `penddemo`)

The most common use of S-functions is to create custom Simulink blocks (see "Creating Custom Blocks" in *Using Simulink*). When you use an S-function to create a general-purpose block, you can use it many times in a model, varying parameters with each instance of the block.

How S-Functions Work

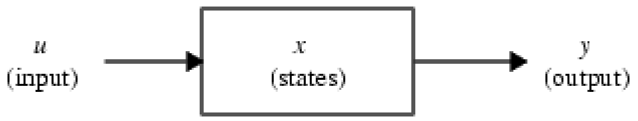
In this section...

Introduction

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

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 simulation time, the inputs, and the states.



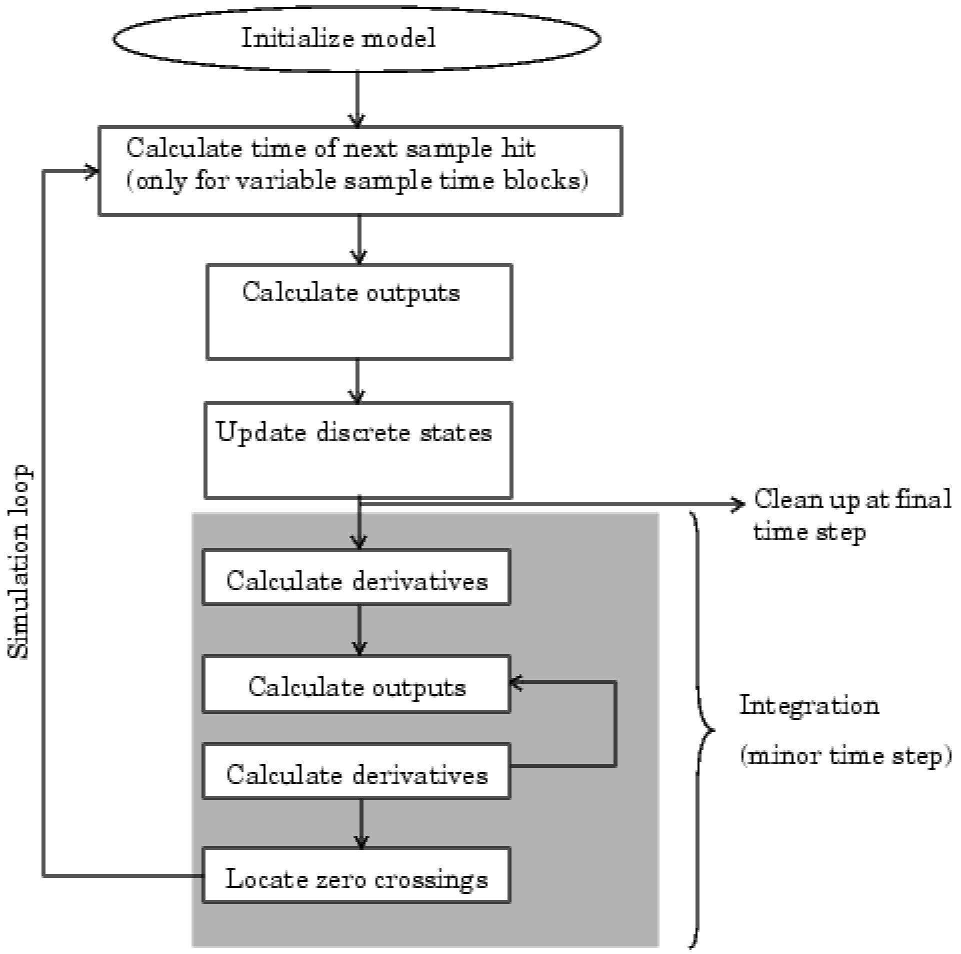
The following equations express the mathematical relationships between the inputs, outputs, states, and simulation time.

$$
\begin{aligned}
&y = f_0(t, x, u) && \text{(Outputs)} \\
&\dot{x}_c = f_d(t, x, u) && \text{(Derivatives)} \\
&x_{d_{k+1}} = f_u(t, x_c, x_{d_k}, u) && \text{(Update)} \\
&\text{where} \quad x = [x_c; x_d]
\end{aligned}
$$

Simulation Stages

Execution of a Simulink model proceeds in stages. First comes the initialization phase. In this phase, the Simulink engine incorporates library blocks into the model, propagates signal widths, data types, and sample times, evaluates block parameters, determines block execution order, and allocates memory. The engine then enters a *simulation loop*, where each pass through the loop is referred to as a *simulation step*. During each simulation step, the engine executes each block in the model in the order determined during initialization. For each block, the engine invokes functions that compute the block states, derivatives, and outputs for the current sample time.

The following figure illustrates the stages of a simulation. The inner integration loop takes place only if the model contains continuous states. The engine executes this loop until the solver reaches the desired accuracy for the state computations. The entire simulation loop then continues until the simulation is complete. See "Simulating Dynamic Systems" in *Using Simulink* for more detailed information on how the engine executes a model. See "How the Simulink Engine Interacts with C S-Functions" on page 4-77 for a description of how the engine calls the S-function API during initialization and simulation.



How the Simulink® Engine 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, the Simulink engine calls the appropriate methods for each S-Function block in the model. Tasks performed by S-function callback methods include:

- Initialization — Prior to the first simulation loop, the engine initializes the S-function, including:
  - Initializing the `SimStruct`, a simulation structure that contains information about the S-function
  - Setting the number and dimensions of input and output ports
  - Setting the block sample times
  - Allocating storage areas
- Calculation of next sample hit — If you 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 block output ports are valid for the current time step.
- Update of discrete states in the major time step — In this call, the block performs once-per-time-step activities such as updating discrete states.
- Integration — This applies to models with continuous states and/or nonsampled zero crossings. If your S-function has continuous states, the engine calls the output and derivative portions of your S-function at minor time steps. This is so the solvers can compute the states for your S-function. If your S-function has nonsampled zero crossings, the engine also 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 *Using Simulink* for an explanation of major and minor time steps.

Implementing S-Functions

In this section...

M-File S-Functions

Level-2 M-file S-functions allow you to create blocks with many of the features and capabilities of Simulink built-in blocks, including:

- Multiple input and output ports
- The ability to accept vector or matrix signals
- Support for various signal attributes including data type, complexity, and signal frames
- Ability to operate at multiple sample rates

A Level-2 M-file S-function consists of a `setup` routine to configure the basic properties of the S-function, and a number of callback methods that the Simulink engine invokes at appropriate times during the simulation.

A basic annotated version of the template resides at

```
matlabroot/toolbox/simulink/blocks/msfuntmpl_basic.m
```

The template consists of a top-level `setup` function and a set of skeleton subfunctions, each of which corresponds to a particular callback method. Each callback method performs a specific S-function task at a particular point in the simulation. The engine invokes the subfunctions using function handles defined in the `setup` routine. See "Level-2 M-File S-Function Callback Methods" on page 3-6 for a table of the supported Level-2 M-file S-function callback methods.

A more detailed Level-2 M-file S-function template resides at

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

We recommend that you follow the structure and naming conventions of the templates when creating Level-2 M-file S-functions. This makes it easier for others to understand and maintain the M-file S-functions that you create. See Chapter 3, "Writing S-Functions in M" for information on creating Level-2 M-file S-functions.

MEX-File S-Functions

Like a Level-2 M-file S-function, a MEX S-function consists of a set of callback methods that the Simulink engine invokes to perform various block-related tasks during a simulation. MEX S-functions can be implemented in C, C++, or Fortran. The engine directly invokes MEX S-function routines instead of using function handles as with M-file S-functions. Because the engine invokes the functions directly, MEX S-functions must follow standard naming conventions specified by the S-function API.

An annotated C MEX-file S-function template resides at

```
matlabroot/simulink/src/sfuntmpl_doc.c
```

The template contains skeleton implementations of all the required and optional callback methods that a C MEX S-function can implement.

For a more basic version of the template see

```
matlabroot/simulink/src/sfuntmpl_basic.c
```

MEX-File Versus M-File S-Functions

Level-2 M-file and MEX-file S-functions each have advantages. The advantage of Level-2 M-file S-functions is speed of development. Developing Level-2 M-file S-functions avoids the time consuming compile-link-execute cycle required when developing in a compiled language. Level-2 M-file S-functions also have easier access to MATLAB toolbox functions and can utilize the MATLAB Editor/Debugger.

MEX-file S-functions are more appropriate for integrating legacy code into a Simulink model. For more complicated systems, MEX-file S-functions may simulate faster than M-file S-functions because the Level-2 M-file S-function calls the MATLAB interpreter for every callback method.

See Chapter 2, “Selecting an S-Function Implementation” for information on choosing the type of S-function best suited for your application.

S-Function Concepts

In this section...

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`) 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`) 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. The Simulink engine integrates the variable $\dot{x}$.

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 (see "Algebraic Loops" in *Using Simulink*). If the simulation results for a model containing your S-function do not converge, or the simulation fails, you may have the direct feedthrough flag incorrectly set. Try turning on the direct feedthrough flag and setting the **Algebraic loop** solver diagnostic to `warning` (see the "Algebraic loop" option on the "Diagnostics Pane: Solver" reference page in *Simulink Graphical User Interface*). Subsequently running the simulation displays any algebraic loops in the model and shows if the engine has placed your S-function within an algebraic loop.

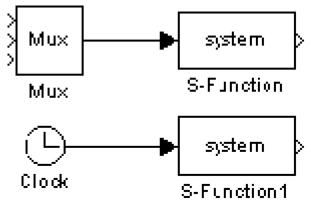
Dynamically Sized Arrays

You can write your S-function to support arbitrary input dimensions. In this case, the Simulink engine determines the actual input dimensions when the simulation is started by evaluating the dimensions of the input vectors driving the S-function. Your S-function can also use the input dimensions to determine the number of continuous states, the number of discrete states, and the number of outputs.

Note A dynamically sized input can have a different size for each instance of the S-function in a particular model or during different simulations, however the input size of each instance of the S-function is static over the course of a particular simulation.

A C MEX S-function and Level-2 M-file S-function can have multiple input and output ports and each port 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. The Simulink engine 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, the engine defines these vectors to be the same length as the input vector.

See "Input and Output Ports" on page 8-19 for more information on configuring S-function input and output ports.

Setting Sample Times and Offsets

Both Level-2 M-file and C MEX S-functions provide the following sample time options, which allow for a high degree of flexibility in specifying when an S-function executes:

- Continuous sample time — For S-functions that have continuous states and/or nonsampled zero crossings (see "How Simulink Works" in *Using Simulink* for an 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 the behavior of your S-function is a function of discrete time intervals, you can define a sample time to control when the Simulink engine calls the S-function. 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 the integer `n` is the current simulation step. The first value of `n` is always zero.

  If you define a discrete sample time, the engine calls the S-function `mdlOutput` and `mdlUpdate` routines at each sample time hit (as defined in the previous 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 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). In this case, you can specify that the sample time is *inherited*. A simple example of this is a Gain block that inherits its sample time from the block driving it.

  An S-function can inherit its sample time from

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

  To specify an S-function sample time is inherited, use -1 in Level-2 M-file S-functions and `INHERITED_SAMPLE_TIME` in C MEX S-functions as the sample time. For more information on the propagation of sample times, see "How Propagation Affects Inherited Sample Times" in the *Simulink User's Guide*.

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`].

Valid C MEX S-Function Sample Times

The valid sample time pairs for a C MEX S-function 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 variable names in 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 C MEX S-function has only one sample time pair, either

```
[INHERITED_SAMPLE_TIME, 0.0]
```

or

```
[INHERITED_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]
```

where

```
INHERITED_SAMPLE_TIME = -1.0
```

Valid Level-2 M-File S-Function Sample Times

The valid sample time pairs for a Level-2 M-file S-function are

```
[0 offset]                                % Continuous sample time
[discrete_sample_time_period, offset] % Discrete sample time
[-1, 0]                                   % Inherited sample time
[-2, 0]                                   % Variable sample time
```

where variable names in italics indicate that a real value is required. When using a continuous sample time, an `offset` of `1` indicates the output is fixed in minor integration time steps. An `offset` of `0` indicates the output changes at every minor integration time step.

Guidelines for Choosing a Sample Time

Use the following guidelines for help with specifying 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]
  ```

  In a C MEX S-function, the `mdlGetTimeOfNextVarHit` routine is called to get the time of the next sample hit for the variable-step discrete task. In a Level-2 M-file S-function, the `NextTimeHit` property is set in the `Outputs` method to set the next sample hit.

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 does not 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 runs at the rate of its driving block, either continuous or discrete, but never runs 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.

See “Sample Times” on page 8-33 for information on implementing different types of sample times in S-functions.

S-Function Examples

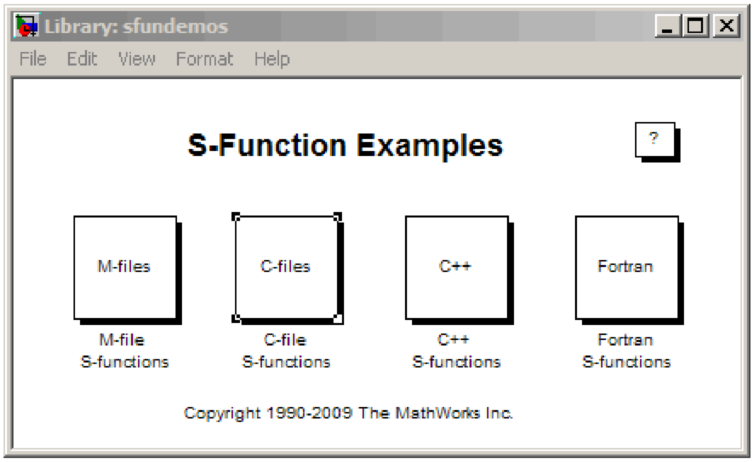
In this section...

Overview of Examples

To run an example:

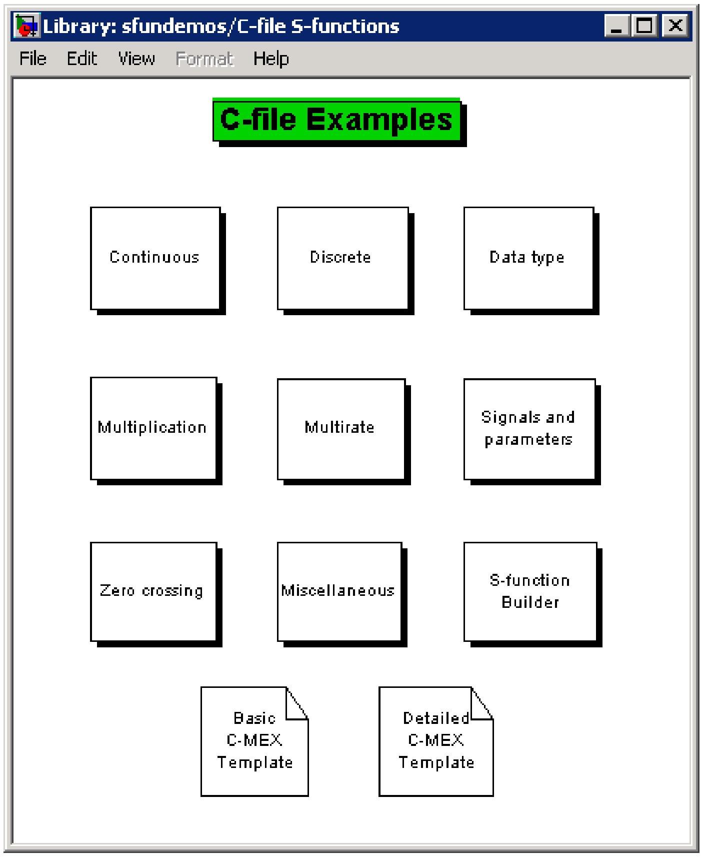
1 Enter `sfundemos` at the MATLAB command prompt.

   The S-function demo library opens.



   Each block represents a category of S-function examples.

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



3 Double-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 the following subdirectories under the MATLAB root directory.

M-files	`toolbox/simulink/simdemos/simfeatures`
C, C++, and Fortran	`toolbox/simulink/simdemos/simfeatures/src`

Level-2 M-File S-Function Examples

The *matlabroot*`/toolbox/simulink/simdemos/simfeatures` directory contains many Level-2 M-file S-functions. Consider starting off by looking at these files.

Filename	Model Name	Description
`msfcn_dsc.m`	`msfcndemo_sfundsc1.mdl`	Implement an S-function with an inherited sample time.
`msfcn_limintm.m`	`msfcndemo_limintm.mdl`	Implement a continuous limited integrator where the output is bounded by lower and upper bounds and includes initial conditions.
`msfcn_multirate.m`	`msfcndemo_multirate.mdl`	Implement a multirate system.
`msfcn_times_two.m`	`msfcndemo_timestwo.mdl`	Implement an S-function that doubles its input.
`msfcn_unit_delay.m`	`msfcndemo_sfundsc2.mdl`	Implement a unit delay.
`msfcn_varpulse.m`	`msfcndemo_varpulse.mdl`	Implement a variable pulse width generator by calling `set_param` from within a Level-2 M-file S-function. Also demonstrates how to use custom set and get methods for the block SimState.
`msfcn_vs.m`	`msfcndemo_vsfunc.mdl`	Implement a variable sample time block in which the first input is delayed by an amount of time determined by the second input.

Level-1 M-File S-Function Examples

The *matlabroot*`/toolbox/simulink/simdemos/simfeatures` directory also contains many Level-1 M-file S-functions, provided as reference for legacy models. Most of these Level-1 M-file S-functions do not have associated demo models.

Filename	Description
`csfunc.m`	Define a continuous system in state-space format.
`dsfunc.m`	Define a discrete system in state-space format.
`limintm.m`	Implement a continuous limited integrator where the output is bounded by lower and upper bounds and includes initial conditions.
`mixedm.m`	Implement a hybrid system consisting of a continuous integrator in series with a unit delay.
`sfun_varargm.m`	Implement an S-function that shows how to use the MATLAB command `varargin`.

`simom.m`	Implement a system in state-space format with internal `A`, `B`, `C`, and `D` matrices, using the equations
	`dx/dt = 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 S-function.

`simom2.m`	Implement a system in state-space format 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 S-function.
`vdlmintm.m`	Implement a discrete limited integrator. This example is identical to `vlimintm.m`, except that the limited integrator is discrete.
`vdpm.m`	Implement the Van der Pol equation (similar to the demo model, `vdp`).
`vlimintm.m`	Implement a continuous limited integrator. This S-function illustrates how to build an S-function that can accommodate a dynamic input and state width.
`vsfunc.m`	Illustrate how to create a variable sample time block. This S-function implements a variable step delay in which the first input is delayed by an amount of time determined by the second input.

C S-Function Examples

The *matlabroot*/toolbox/simulink/simdemos/simfeatures/src directory contains examples of C MEX S-functions, many of which have an M-file S-function counterpart. The C MEX S-functions are listed in the following table.

Filename	Model Name	Description
barplot.c	sfcndemo_barplot.mdl	Access Simulink signals without using the standard block inputs.
csfunc.c	sfcndemo_csfunc.mdl	Implement a continuous system.
dlimintc.c	No model available	Implement a discrete-time limited integrator.
dsfunc.c	sfcndemo_dsfunc.mdl	Implement a discrete system.
limintc.c	No model available	Implement a limited integrator.
mixedm.c	sfcndemo_mixedm.mdl	Implement a hybrid dynamic system consisting of a continuous integrator (1/s) in series with a unit delay (1/z).
mixedmex.c	sfcndemo_mixedmex.mdl	Implement a hybrid dynamic system with a single output and two inputs.
quantize.c	sfcndemo_sfun_quantize.mdl	Implement a vectorized quantizer. Quantizes the input into steps as specified by the quantization interval parameter, q.
sdotproduct.c	sfcndemo_sdotproduct.mdl	Compute dot product (multiply-accumulate) of two real or complex vectors.
sfbuilder_bususage.c	sfbuilder_bususage.mdl	Access S-Function Builder with a bus input and output.
sftable2.c	No model available	Implement a two-dimensional table lookup.
sfun_atol.c	sfcndemo_sfun_atol.mdl	Set different absolute tolerances for each continuous state.

Filename	Model Name	Description
`sfun_cplx.c`	`sfcndemo_cplx.mdl`	Add complex data for an S-function with one input port and one parameter.
`sfun_directlook.c`	No model available	Implement a direct 1-D lookup.
`sfun_dtype_io.c`	`sfcndemo_dtype_io.mdl`	Implement an S-function that uses Simulink data types for inputs and outputs.
`sfun_dtype_param.c`	`sfcndemo_dtype_param.mdl`	Implement an S-function that uses Simulink data types for parameters.
`sfun_dynsize.c`	`sfcndemo_sfun_dynsize.mdl`	Implements dynamically-sized outputs .
`sfun_errhdl.c`	`sfcndemo_sfun_errhdl.mdl`	Check parameters using the `mdlCheckParameters` S-function routine.
`sfun_fcncall.c`	`sfcndemo_sfun_fcncall.mdl`	Execute function-call subsystems on the first and second output elements.
`sfun_frmad.c`	`sfcndemo_frame.mdl`	Implement a frame-based A/D converter.
`sfun_frmda.c`	`sfcndemo_frame.mdl`	Implement a frame-based D/A converter.
`sfun_frmdft.c`	`sfcndemo_frame.mdl`	Implement a multichannel frame-based Discrete-Fourier transformation (and its inverse).
`sfun_frmunbuff.c`	`sfcndemo_frame.mdl`	Implement a frame-based unbuffer block.
`sfun_multiport.c`	`sfcndemo_sfun_multiport.mdl`	Configure multiple input and output ports.
`sfun_manswitch.c`	No model available	Implement a manual switch.
`sfun_matadd.c`	`sfcndemo_matadd.mdl`	Add matrices in an S-function with one input port, one output port, and one parameter.

Filename	Model Name	Description
sfun_multirate.c	sfcndemo_sfun_multirate.mdl	Demonstrate how to specify port-based sample times.
sfun_port_constant.c	sfcndemo_port_constant.mdl	Demonstrate how to specify constant port-based sample times.
sfun_port_triggered.c	sfcndemo_port_triggered.mdl	Demonstrate how to use port-based sample times in a triggered subsystem.
sfun_runtime1.c	sfcndemo_runtime.mdl	Implement run-time parameters for all tunable parameters.
sfun_runtime2.c	sfcndemo_runtime.mdl	Register individual run-time parameters.
sfun_runtime3.c	sfcndemo_runtime.mdl	Register dialog parameters as run-time parameters.
sfun_runtime4.c	sfcndemo_runtime.mdl	Implement run-time parameters as a function of multiple dialog parameters.
sfun_simstate.c	sfcndemo_sfun_simstate.mdl	Demonstrate the S-function API for saving and restoring the SimState.
sfun_zc.c	sfcndemo_sfun_simstate.mdl	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	sfcndemo_sfun_zc_sat.mdl	Demonstrate zero crossings with saturation.
sfunmem.c	sfcndemo_sfunmem.mdl	Implement a one-integration-step delay and hold memory function.

Filename	Model Name	Description

`simomex.c`	`sfcndemo_simomex.mdl`	Implement a single-input, two-output state-space dynamic system described by the 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.

`stspace.c`	`sfcndemo_stspace.mdl`	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.
`stvctf.c`	`sfcndemo_stvctf.mdl`	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.
`stvdtf.c`	`sfcndemo_stvdtf.mdl`	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`	`sfcndemo_stvmgain.mdl`	Implement a time-varying matrix gain.

Filename	Model Name	Description
`table3.c`	No model available	Implement a 3-D lookup table.
`timestwo.c`	`sfcndemo_timestwo.mdl`	Implement a C MEX S-function that doubles its input.
`vdlmintc.c`	No model available	Implement a discrete-time vectorized limited integrator.
`vdpmex.c`	`sfcndemo_vdpmex.mdl`	Implement the Van der Pol equation.
`vlimintc.c`	No model available	Implement a vectorized limited integrator.
`vsfunc.c`	`sfcndemo_vsfunc.mdl`	Illustrate 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.

Fortran S-Function Examples

The following table lists sample Fortran S-functions available in the *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src` directory.

Filename	Model Name	Description
`sfun_timestwo_for.F`	`sfcndemo_timestwo_for.mdl`	Implement a Level-1 Fortran S-function that represents the `timestwo.c` S-function.
`sfun_atmos.c` `sfun_atmos_sub.F`	`sfcndemo_atmos.mdl`	Calculate the 1976 standard atmosphere to 86 km using a Fortran subroutine.
`simomexf.F`	No model available	Implement a Level-1 Fortran S-function that represents the `simomex.c` S-function.
`vdpmexf.F`	No model available	Implement a Level-1 Fortran S-function that represents the `vdpmex.c` S-function.

C++ S-Function Examples

The following table lists sample C++ S-functions available in the `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src` directory.

Filename	Model Name	Description
`sfun_counter_cpp.cpp`	`sfcndemo_counter_cpp.mdl`	Store a C++ object in the pointers vector `PWork`.

Selecting an S-Function Implementation

Available S-Function Implementations

You can implement your S-function in one of five ways:

- **A Level-1 M-file S-function** provides a simple M interface to interact with a small portion of the S-function API. Level-2 M-file S-functions supersede Level-1 M-file S-functions.

- **A Level-2 M-file S-function** provides access to a more extensive set of the S-function API and supports code generation. In most cases, use a Level-2 M-file S-function when you want to implement your S-function in M.

- **A handwritten C MEX S-function** provides the most programming flexibility. You can implement your algorithm as a C MEX S-function or write a wrapper S-function to call existing C, C++, or Fortran code. Writing a new S-function requires knowledge of the S-function API and, if you want to generate inlined code for the S-function, the Target Language Compiler (TLC).
- **The S-Function Builder** is a graphical user interface for programming a subset of S-function functionality. If you are new to writing C MEX S-functions, you can use the S-Function Builder to generate new S-functions or incorporate existing C or C++ code without interacting with the S-function API. The S-Function Builder can also generate TLC files for inlining your S-function during code generation with the Real-Time Workshop product.
- **The Legacy Code Tool** is a set of MATLAB commands that helps you create an S-function to incorporate legacy C or C++ code. Like the S-Function Builder, the Legacy Code Tool can generate a TLC file to inline your S-function during code generation. The Legacy Code Tool provides access to fewer of the methods in the S-function API than the S-Function Builder or a handwritten C MEX S-function.

The following sections describe the uses, features, and differences of these S-function implementations. The last section compares using a handwritten C MEX S-function, the S-Function Builder, and the Legacy Code Tool to incorporate an existing C function into your Simulink model.

What Type of S-Function Should You Use?

Consider the following questions if you are unclear about what type of S-function is best for your application.

If you...	Then use...
Are an M programmer with little or no C programming experience	A Level-2 M-file S-function, especially if you do not need to generate code for a model containing the S-function (see Chapter 3, "Writing S-Functions in M").
Need to generate code for a model containing the S-function	Either a Level-2 M-file S-function or a C MEX S-functions. Level-2 M-file S-functions require that you write a Target Language Compiler (TLC) file for your S-function, before generating code. C MEX S-functions, however, automatically support code generation.
Need the simulation to run faster	A C MEX S-function, even if you do not need to generate code (see Chapter 4, "Writing S-Functions in C"). For complicated systems, Level-2 M-file S-functions simulate slower than C MEX S-functions because they call out to the MATLAB interpreter.
Need to implement the S-function in C, but have no previous experience writing C MEX S-functions	The S-Function Builder.

If you...	Then use...
Are incorporating legacy code into the model	Any S-function, with the exception of a Level-1 M-file S-function. Consider using the Legacy Code Tool if your legacy function calculates only outputs, not dynamic states (see “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” on page 4-55). Otherwise, consider using the S-Function Builder. If you need to call the legacy code during simulation, do not use a Level-2 M-file S-function because they call legacy code only through their TLC files.
Need to generate embeddable code for an S-function that incorporates legacy code	The Legacy Code Tool if your legacy function calculates only outputs. Otherwise, use a handwritten C MEX S-function or the S-Function Builder.

How to Implement S-Functions

The following table gives an overview of how to write different types of S-functions. See the associated sections of the S-function documentation for more details on how to implement S-functions using a particular method.

Note For backward compatibility, the following table and sections contain information about Level-1 M-file S-functions. However, use the Level-2 M-file S-function API to develop new M-file S-functions.

S-Function Type	Implementation
Level-1 M-file S-function	Use the following template to write a new Level-1 M-file S-function: `sfuntmpl.m` See “Maintaining Level-1 M-File S-Functions” on page 3-14 for more information.
Level-2 M-file S-function	**1** Use the following template to write a new Level-2 M-file S-function: `msfuntmpl_basic.m` See “Writing Level-2 M-File S-Functions” on page 3-4 for more information. **2** Write a Target Language Compiler (TLC) file for the S-function if you need to generate code for a model containing the S-function. The file, `msfcn_times_two.tlc`in the directory is an example TLC file for the S-function `msfcn_times_two.m`. See “Inlining M-File S-Functions” in *Real-Time Workshop Target Language Compiler* for information on writing TLC files for Level-2 M-file S-functions.

S-Function Type	Implementation
Hand-written C MEX S-function	**1** Use the following template to write a new C MEX S-function (see “Example of a Basic C MEX S-Function” on page 4-43) or to write a wrapper S-function that calls C, C++, or Fortran code: `matlabroot/simulink/src/sfuntmpl_doc.c` See “Writing Wrapper S-Functions” in the *Real-Time Workshop User’s Guide* for information on writing wrapper S-functions to incorporate legacy C or C++ code. See “Constructing the Gateway” on page 6-13 for information on writing a wrapper function to incorporate legacy Fortran code. **2** Compile the S-function using the `mex` command to obtain an executable to use during simulation. **3** Write a TLC file for the S-function if you want to inline the code during code generation (see “Writing Fully Inlined S-Functions with the mdlRTW Routine” in the *Real-Time Workshop User’s Guide* and *Real-Time Workshop Target Language Compiler*). You do not need a TLC file if you are not inlining the S-function in the generated code.

S-Function Type	Implementation
S-Function Builder	**1** Enter the S-function attributes into the S-Function Builder dialog box (see “S-Function Builder Dialog Box” on page 4-12). **2** Select the **Generate wrapper TLC** option to generate a TLC file to inline the S-function during code generation. **3** Click **Build** to generate the S-function, TLC file, and an executable file to use during simulation.

Legacy Code Tool	Use the `legacy_code` function to perform the following steps (see “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” on page 4-55):
	
	1 Initialize a data structure that describes the S-function attributes in terms of the legacy function.
	
	`lct_spec = legacy_code('initialize');`
	
	2 Generate and compile the wrapper S-function.
	
	`legacy_code('sfcn_cmex_generate', lct_spec);`
	`legacy_code('compile', lct_spec);`
	
	3 Instantiate an S-Function block that calls the S-function wrapper.
	
	`legacy_code('slblock_generate', lct_spec);`
	
	4 Generate a TLC file to inline the S-function during code generation.
	
	`legacy_code('sfcn_tlc_generate', lct_spec);`

S-Function Features

The following tables give overviews of the features supported by different types of S-functions. The first table focuses on handwritten S-functions. The second table compares the features of S-functions automatically generated by the S-Function Builder or Legacy Code Tool.

Features of Hand-written S-Functions

Feature	Level-1 M-File S-Function	Level-2 M-File S-Function	Handwritten C MEX S-Function
Data types	Supports signals with a data type of `double`.	Supports any data type supported by Simulink software, including fixed-point types.	Supports any data type supported by Simulink software, including fixed-point types.
Numeric types	Supports only real signals.	Supports real and complex signals.	Supports real and complex signals.
Frame support	Does not support frame-based signals.	Supports framed and unframed signals.	Supports framed and unframed signals.
Port dimensions	Supports vector inputs and outputs. Does not support multiple input and output ports.	Supports scalar, 1-D, and multidimensional input and output signals.	Supports scalar, 1-D, and multidimensional input and output signals.
S-function API	Supports only `mdlInitializeSizes`, `mdlDerivatives`, `mdlUpdate`, `mdlOutputs`, `mdlGetTimeOfNextVarHit`, and `mdlTerminate`.	Supports a larger set of the S-function API. See "Level-2 M-File S-Function Callback Methods" on page 3-6 for a list of supported methods.	Supports the entire S-function API.
Code generation support	Does not support code generation.	Requires a handwritten TLC file to generate code.	Natively supports code generation. Requires a handwritten TLC file to inline the S-function during code generation.

Features of Hand-written S-Functions (Continued)

Feature	Level-1 M-File S-Function	Level-2 M-File S-Function	Handwritten C MEX S-Function
Simulink® Accelerator™ mode	Runs interpretively and is, therefore, not accelerated.	Provides the option to use a TLC file in Accelerator mode, instead of running interpretively.	Provides the option to use a TLC or MEX-file in Accelerator mode.
Model reference	Cannot be used in a referenced model.	Supports Normal and Accelerator mode simulations when used in a referenced model. Requires a TLC file for Accelerator mode.	Provides options for sample time inheritance and Normal mode support when used in a referenced model.
`Simulink.AliasType`, `Simulink.NumericType` and `Simulink.StructType` support	Does not support these classes.	Supports `Simulink.NumericType` and `Simulink.AliasType` classes (see "Custom Data Types" on page 8-29).	Supports all of these classes (see "Custom Data Types" on page 8-29). However, supports `Simulink.StructType` only for S-function parameters.
Bus input and output signals	Does not support bus input or output signals.	Does not support bus input or output signals.	Does not support bus input or output signals.
Tunable and run-time parameters	Supports tunable parameters during simulation. Does not support run-time parameters.	Supports tunable and run-time parameters.	Supports tunable and run-time parameters.
Work vectors	Does not support work vectors.	Supports DWork vectors (see "Using DWork Vectors in Level-2 M-File S-Functions" on page 7-12).	Supports all work vector types (see Chapter 7, "Using Work Vectors").

Features of Automatically Generated S-Functions

Feature	S-Function Builder	Legacy Code Tool
Data types	Supports any data type supported by Simulink software, including fixed-point types.	Supports all built-in data types. To use a fixed-point data type, you must specify the data type as a `Simulink.NumericType`. You cannot use a fixed-point type with unspecified scaling.
Numeric types	Supports real and complex signals.	Supports complex signals only for built-in data types.
Frame support	Supports framed and unframed signals.	Does not support frame-based signals.
Port dimensions	Supports scalar, 1-D, and multidimensional input and output signals.	Supports scalar, 1-D, and multidimensional input and output signals.
S-function API	Supports `mdlInitializeSizes`, `mdlInitializeSampleTimes`, `mdlStart`, `mdlDerivative`, `mdlUpdate`, `mdlOutput`, and `mdlTerminate`.	Supports `mdlInitializeSizes`, `mdlInitializeSampleTimes`, `mdlStart`, `mdlInitializeConditions`, `mdlOutputs`, and `mdlTerminate`.
Code generation support	Natively supports code generation. Also, automatically generates a TLC file for inlining the S-function during code generation.	Natively supports code generation optimized for embedded systems. Also, automatically generates a TLC file that supports expression folding for inlining the S-function during code generation.
Simulink Accelerator mode	Uses a TLC file in Accelerator mode, if the file was generated. Otherwise, uses the MEX-file.	Provides the option to use a TLC or MEX-file in Accelerator mode.
Model reference	Uses default behaviors when used in a referenced model.	Uses default behaviors when used in a referenced model.

Features of Automatically Generated S-Functions (Continued)

Feature	S-Function Builder	Legacy Code Tool
`Simulink.AliasType`, `Simulink.NumericType`, and `Simulink.StructType`	Does not support these classes.	Supports `Simulink.AliasType` and `Simulink.NumericType`.
Bus input and output signals	Does not support bus input or output signals.	Supports bus input and output signals. You must define a `Simulink.Bus` object in the MATLAB workspace that is equivalent to the structure of the input or output used in the legacy code. Does not support bus parameters.
Tunable and run-time parameters	Supports tunable parameters only during simulation. Supports run-time parameters.	Supports tunable and run-time parameters.
Work vectors	Does not provide access to work vectors.	Supports DWork vectors with the usage type `SS_DWORK_USED_AS_DWORK`. See "Types of DWork Vectors" on page 7-5 for a discussion on the different DWork vector usage types.

S-Function Limitations

The following table summarizes the major limitations of the different types of S-functions.

Implementation	Limitations
Level-1 M-file S-function	Does not support the majority of S-function features. See the "S-Function Features" on page 2-8 section for information on what features a Level-1 M-file S-function does support.
Level-2 M-file S-functions	• Does not support bus input and output signals. • Cannot incorporate legacy code during simulation, only during code generation through a TLC file.
Handwritten C MEX S-function	• Does not support bus input and output signals. • Does not support model referencing under all circumstances. See "Simulink Model Referencing Limitations" in *Using Simulink* for details.

Implementation	Limitations
S-Function Builder	• Generates S-function code using a wrapper function which incurs additional overhead. • Does not support the following S-function features: - Work vectors - Port-based sample times - Multiple sample times or a nonzero offset time - Dynamically-sized input and output signals for an S-function with multiple input and output ports **Note** S-functions with one input and one output port can have dynamically-sized signals
Legacy Code Tool	• Generates C MEX S-functions for existing functions written in C or C++ only. The tool does not support transformation of MATLAB or Fortran functions. • Can interface with C++ functions, but not C++ objects. • Does not support simulating continuous or discrete states. • Does not support use of function pointers as the output of the legacy function being called. • Always sets the S-function's flag for direct feedthrough (`sizes.DirFeedthrough`) to `true`. • Supports only the continuous, but fixed in minor time step, sample time and offset option. • Supports complex numbers, but only with Simulink built-in data types. • Does not support the following S-function features: - Work vectors, other then general DWork vectors - Frame-based input and output signals - Port-based sample times - Multiple block-based sample times

Example Using S-Functions to Incorporate Legacy C Code

In this section...

Overview

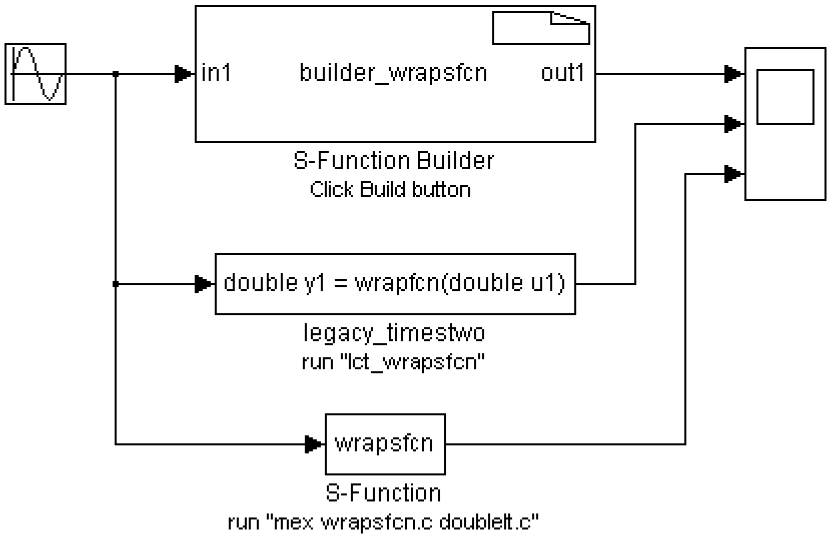
C MEX S-functions allow you to call existing C code within your Simulink models. For example, consider the simple C function `doubleIt.c` that outputs a value two times the value of the function input.

```
double doubleIt(double u)
{
    return(u * 2.0);
}
```

You can create an S-function that calls `doubleIt.c` by either:

- Writing a wrapper S-function. Using this method, you hand write a new C S-function and associated TLC file. This method requires the most knowledge about the structure of a C S-function.
- Using an S-Function Builder block. Using this method, you enter the characteristics of the S-function into a block dialog. This method does not require any knowledge about writing S-functions. However, a basic understanding of the structure of an S-function can make the S-Function Builder dialog box easier to use.
- Using the Legacy Code Tool. Using this command line method, you define the characteristics of your S-function in a data structure in the MATLAB workspace. This method requires the least amount of knowledge about S-functions.

The following sections describe how to create S-functions for use in a Simulink simulation and with Real-Time Workshop code generation, using the previous three methods. The model `sfcndemo_choosing_sfun.mdl` contains blocks that use these S-functions. Copy this model and the files `doubleIt.c` and `doubleIt.h` from the directory *docroot*`/toolbox/simulink/sfg/examples` into your working directory if you plan to step through the examples.



Using a Hand-Written S-Function to Incorporate Legacy Code

The S-function `wrapsfcn.c` calls the legacy function `doubleIt.c` in its `mdlOutputs` method. Save the `wrapsfcn.c` file into your working directory, if you are planning to compile the S-function to run in the example model `sfcndemo_choosing_sfun.mdl`.

To incorporate the legacy code into the S-function, `wrapsfcn.c` begins by declaring `doubleIt.c` with the following line:

```
extern real_T doubleIt(real_T u);
```

Once declared, the S-function can use `doubleIt.c` in its `mdlOutputs` method. For example:

```
/* Function: mdlOutputs =======================================
 * Abstract:
 *    Calls the doubleIt.c function to multiple the input by 2.
 */
static void mdlOutputs(SimStruct *S, int tid){
  InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
  real_T            *y    = ssGetOutputPortRealSignal(S,0);

  *y = doubleIt(*uPtrs[0]);
}
```

To compile the `wrapsfcn.c` S-function, run the following `mex` command. Make sure that the `doubleIt.c` file is in your working directory.

```
mex wrapsfcn.c doubleIt.c
```

To generate code for the S-function using the Real-Time Workshop code generator, you need to write a Target Language Compiler (TLC) file. The following TLC file `wrapsfcn.tlc` uses the `BlockTypeSetup` function to declare a function prototype for `doubleIt.c`. The TLC file's `Outputs` function then tells the Real-Time Workshop code generator how to inline the call to `doubleIt.c`. For example:

```
%implements "wrapsfcn" "C"
%% File    : wrapsfcn.tlc
%% Abstract:
%%      Example tlc file for S-function wrapsfcn.c
%%

%% Function: BlockTypeSetup ================================
%% Abstract:
%%      Create function prototype in model.h as:
%%     "extern double doubleIt(double u);"
%%

%function BlockTypeSetup(block, system) void
  %openfile buffer

  %% PROVIDE ONE LINE OF CODE AS A FUNCTION PROTOTYPE
  extern double doubleIt(double u);

  %closefile buffer
  %<LibCacheFunctionPrototype(buffer)>
  %%endfunction %% BlockTypeSetup

%% Function: Outputs ======================================
%% Abstract:
%%      CALL LEGACY FUNCTION: y = doubleIt( u );
%%

%function Outputs(block, system) Output

  /* %<Type> Block: %<Name> */

  %assign u = LibBlockInputSignal(0, "", "", 0)
  %assign y = LibBlockOutputSignal(0, "", "", 0)

  %% PROVIDE THE CALLING STATEMENT FOR "doubleIt"
  %<y> = doubleIt( %<u> );

%endfunction %% Outputs
```

See *Real-Time Workshop Target Language Compiler* for more information on writing TLC files.

Using the S-Function Builder to Incorporate Legacy Code

The S-Function Builder automates the creation of S-functions and TLC files that incorporate legacy code. For this example, in addition to `doubleIt.c`, you need the header file `doubleIt.h` that declares the `doubleIt.c` function format, as follows:

```
extern real_T doubleIt(real_T in1);
```

The S-Function Builder block in `sfcndemo_choosing_sfun.mdl` shows how to configure the block dialog to call the legacy function `doubleIt.c`. In the S-Function Builder block dialog:

- The **S-function name** field in the **Parameters** pane defines the name `builder_wrapsfcn` for the generated S-function.

- The **Data Properties** pane names the input and output ports as `in1` and `out1`, respectively.
- The **Libraries** pane provides the interface to the legacy code.
  - The **Library/Object/Source files** field contains the source file name `doubleIt.c`.
  - The **Includes** field contains the following line to include the header file that declares the legacy function:

    ```
    #include <doubleIt.h>
    ```

- The **Outputs** pane calls the legacy function with the lines:

  ```
  /* Call function that multiplies the input by 2 */

       *out1 = doubleIt(*in1);
  ```

- The **Build Info** pane selects the **Generate wrapper TLC** option.

When you click **Build**, the S-Function Builder generates three files.

File Name	Description
`builder_wrapsfcn.c`	The main S-function.
`builder_wrapsfcn_wrapper.c`	A wrapper file containing separate functions for the code entered in the **Outputs**, **Continuous Derivatives**, and **Discrete Updates** panes of the S-Function Builder.
`builder_wrapsfcn.tlc`	The S-function's TLC file.

The `builder_wrapsfcn.c` file follows a standard format:

- The file begins with a set of `#define` statements that incorporate the information from the S-Function Builder. For example, the following lines define the first input port:

  ```
  #define NUM_INPUTS          1
  /* Input Port  0 */
  #define IN_PORT_0_NAME      in1
  ```

```
#define INPUT_0_WIDTH       1
#define INPUT_DIMS_0_COL    1
#define INPUT_0_DTYPE       real_T
#define INPUT_0_COMPLEX     COMPLEX_NO
#define IN_0_FRAME_BASED    FRAME_NO
#define IN_0_DIMS           1-D
#define INPUT_0_FEEDTHROUGH 1
```

- Next, the file declares all the wrapper functions found in the `builder_wrapsfcn_wrapper.c` file. This example requires only a wrapper function for the **Outputs** code.

```
extern void builder_wrapsfcn_Outputs_wrapper(const real_T *in1,
                          real_T *out1);
```

- Following these definitions and declarations, the file contains the S-function methods, such as `mdlInitializeSizes`, that initialize the S-function's input ports, output ports, and parameters. See "Process View" on page 4-77 for a list of methods that are called during the S-function initialization phase.
- The file's `mdlOutputs` method calls the `builder_wrapsfcn_wrapper.c` function. The method uses the input and output names `in1` and `out1`, as defined in the **Data Properties** pane, when calling the wrapper function. For example:

```
/* Function: mdlOutputs =============================================
 *
*/
static void mdlOutputs(SimStruct *S, int_T tid)
{
    const real_T   *in1  = (const real_T*) ssGetInputPortSignal(S,0);
    real_T        *out1  = (real_T *)ssGetOutputPortRealSignal(S,0);

    builder_wrapsfcn_Outputs_wrapper(in1, out1);
}
```

- The file `builder_wrapsfcn.c` concludes with the required `mdlTerminate` method.

The wrapper function `builder_wrapsfcn_wrapper.c` has three parts:

- The `Include Files` section includes the `doubleIt.h` file, along with the standard S-function header files:

```
/*
 * Include Files
 *
 */
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
#include <math.h>
#include <doubleIt.h>
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
```

- The `External References` section contains information from the **External reference declarations** field on the **Libraries** pane. This example does not use this section.

- The `Output functions` section declares the function `builder_wrapfcn_Outputs_wrapper`, which contains the code entered in the S-Function Builder block dialog's **Outputs** pane:

```
/*
 * Output functions
 *
 */
void builder_wrapfcn_Outputs_wrapper(const real_T *in1,
                          real_T *out1)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
/* Call function that multiplies the input by 2 */

      *out1 = doubleIt(*in1);
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_END --- EDIT HERE TO _BEGIN */
}
```

Note Compared to a handwritten S-function, the S-Function Builder places the call to the legacy C function down an additional level through the wrapper file `builder_wrapsfcn_wrapper.c`.

The TLC file `builder_wrapsfcn.tlc` generated by the S-Function Builder is similar to the previous handwritten version. The file declares the legacy function in `BlockTypeSetup` and calls it in the `Outputs` method.

```
%implements  builder_wrapsfcn "C"
%% Function: BlockTypeSetup ====================================
%%
%% Purpose:
%%      Set up external references for wrapper functions in the
%%      generated code.
%%
%function BlockTypeSetup(block, system) Output
 %openfile externs

 extern void builder_wrapsfcn_Outputs_wrapper(const real_T *in1,
                          real_T *out1);
 %closefile externs
 %<LibCacheExtern(externs)>
 %%
%endfunction

%% Function: Outputs ===========================================
%%
%% Purpose:
%%      Code generation rules for mdlOutputs function.
%%
%function Outputs(block, system) Output
   /* S-Function "builder_wrapsfcn_wrapper" Block: %<Name> */

 %assign pu0 = LibBlockInputSignalAddr(0, "", "", 0)
 %assign py0 = LibBlockOutputSignalAddr(0, "", "", 0)
 %assign py_width = LibBlockOutputSignalWidth(0)
 %assign pu_width = LibBlockInputSignalWidth(0)
 builder_wrapsfcn_Outputs_wrapper(%<pu0>, %<py0> );
 %%
%endfunction
```

Using the Legacy Code Tool to Incorporate Legacy Code

The section “Example of Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” on page 4-58 in “Writing S-Functions in C” shows how to use the Legacy Code Tool to create an S-function that incorporates `doubleIt.c`. For a script that performs the steps in that example, copy the file `lct_wrapsfcn.m` to your working directory. Make sure that the `doubleIt.c` and `doubleIt.h` files are in your working directory then run the script by typing `lct_wrapsfcn` at the MATLAB command prompt. The script creates and compiles the S-function `legacy_wrapsfcn.c` and creates the TLC file `legacy_wrapsfcn.tlc` via the following commands.

```
% Create the data structure
def = legacy_code('initialize');

% Populate the data struture
def.SourceFiles = {'doubleIt.c'};
def.HeaderFiles = {'doubleIt.h'};
def.SFunctionName = 'legacy_wrapsfcn';
def.OutputFcnSpec = 'double y1 = doubleIt(double u1)';
def.SampleTime = [-1,0];

% Generate the S-function
legacy_code('sfcn_cmex_generate', def);

% Compile the MEX-file
legacy_code('compile', def);

% Generate a TLC-file
legacy_code('sfcn_tlc_generate', def);
```

The S-function `legacy_wrapsfcn.c` generated by the Legacy Code Tool begins by including the `doubleIt.h` header file. The `mdlOutputs` method then directly calls the `doubleIt.c` function, as follows:

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

```
{
  /*
   * Get access to Parameter/Input/Output/DWork/size information
   */
  real_T *u1 = (real_T *) ssGetInputPortSignal(S, 0);
  real_T *y1 = (real_T *) ssGetOutputPortSignal(S, 0);

  /*
   * Call the legacy code function
   */
  *y1 = doubleIt( *u1);
}
```

The S-function generated by the Legacy Code Tool differs from the S-function generated by the S-Function Builder as follows:

- The S-function generated by the S-Function Builder calls the legacy function `doubleIt.c` through the wrapper function `builder_wrapsfcn_wrapper.c`. The S-function generated by the Legacy Code Tool directly calls `doubleIt.c` from its `mdlOutputs` method.
- The S-Function Builder uses the input and output names entered into the **Data Properties** pane, allowing you to customize these names in the S-function. The Legacy Code Tool uses the default names `y` and `u` for the outputs and inputs, respectively. You cannot specify customized names to use in the generated S-function when using the Legacy Code Tool.
- The S-Function Builder and Legacy Code Tool both specify an inherited sample time, by default. However, the S-Function Builder uses an offset time of `0.0` while the Legacy Code Tool specifies that the offset time is fixed in minor time steps.

The TLC file `legacy_wrapsfcn.tlc` supports expression folding by defining `BlockInstanceSetup` and `BlockOutputSignal` functions. The TLC file also contains a `BlockTypeSetup` function to declare a function prototype for `doubleIt.c` and an `Outputs` function to tell the Real-Time Workshop code generator how to inline the call to `doubleIt.c`.:

```
%% Function: BlockTypeSetup ===============================================
%%
%function BlockTypeSetup(block, system) void
```

```
  %%
  %% The Target Language must be C
  %if ::GenCPP==1
    %<LibReportFatalError("This S-Function generated by the Legacy Code Tool
         must be only used with the C Target Language")>
  %endif
  %<LibAddToCommonIncludes("doubleIt.h")>
  %<LibAddToModelSources("doubleIt")>
%%
%endfunction

%% Function: BlockInstanceSetup ===========================================
%%
%function BlockInstanceSetup(block, system) void
  %%
  %<LibBlockSetIsExpressionCompliant(block)>
  %%
%endfunction

%% Function: Outputs ======================================================
%%
%function Outputs(block, system) Output
  %%
    %if !LibBlockOutputSignalIsExpr(0)
    %assign u1_val = LibBlockInputSignal(0, "", "", 0)
    %assign y1_val = LibBlockOutputSignal(0, "", "", 0)
    %%
%<y1_val = doubleIt( %<u1_val>);
    %endif
  %%
%endfunction

%% Function: BlockOutputSignal ============================================
%%
%function BlockOutputSignal(block,system,portIdx,ucv,lcv,idx,retType) void
  %%
  %assign u1_val = LibBlockInputSignal(0, "", "", 0)
  %assign y1_val = LibBlockOutputSignal(0, "", "", 0)
  %%
  %switch retType
```

```
    %case "Signal"
      %if portIdx == 0
        %return "doubleIt( %<u1_val>)"
      %else
        %assign errTxt = "Block output port index not supported: %<portIdx>"
%endif
    %default
      %assign errTxt = "Unsupported return type: %<retType>"
      %<LibBlockReportError(block,errTxt)>
  %endswitch
```

Writing S-Functions in M

Introduction

You can create custom blocks whose properties and behaviors are defined by M-file functions called M-file S-functions. The Level-2 M-file S-function application programming interface (API) allows you to create blocks that have many of the features and capabilities of Simulink built-in blocks, including:

- Multiple input and output ports
- 1-D, 2-D, and n-D input and output signals
- All data types supported by the Simulink software
- Real or complex signals
- Frame-based signals
- Multiple sample rates
- User-defined data and work vectors
- Tunable and run-time parameters

Note Level-2 M-file S-functions do not support zero-crossing detection.

For information on how to write a Level-2 M-file S-functions, see “Writing Level-2 M-File S-Functions” on page 3-4.

You can generate code for Level-2 M-file S-functions if they are inlined. See “Inlining S-Functions” in the Real-Time Workshop Target Language Compiler documentation for more information.

Note This version of the Simulink software also supports a predecessor API known as the Level-1 M-file S-function. This ensures that you can simulate models developed with earlier releases that use Level-1 M-file S-functions in their S-Function blocks (see "Maintaining Level-1 M-File S-Functions" on page 3-14). Level-1 M-file S-functions support a much smaller subset of the S-function API then Level-2 M-file S-functions, and their features are limited compared to built-in blocks. Use the Level-2 API, not the Level-1 API, to develop new M-file S-functions.

Writing Level-2 M-File S-Functions

In this section...

About Level-2 M-File S-Functions

The Level-2 M-file S-function API allows you to use the MATLAB M language to create custom blocks with multiple input and output ports and capable of handling any type of signal produced by a Simulink model, including matrix and frame signals of any data type. The Level-2 M-file S-function API corresponds closely to the API for creating C MEX-file S-functions. Much of the documentation for creating C MEX-file S-functions (see Chapter 4, "Writing S-Functions in C" and Chapter 8, "Implementing Block Features") applies also to Level-2 M-file S-functions. To avoid duplication, this section focuses on providing information that is specific to writing Level-2 M-file S-functions.

A Level-2 M-file S-function is an M-file that defines the properties and behavior of an instance of a Level-2 M-File S-Function block that references the M-file in a Simulink model. The M-file itself comprises a set of callback methods (see "Level-2 M-File S-Function Callback Methods" on page 3-6) that the Simulink engine invokes when updating or simulating the model. The callback methods perform the actual work of initializing and computing the outputs of the block defined by the S-function.

To facilitate these tasks, the engine passes a run-time object to the callback methods as an argument. The run-time object effectively serves as an M proxy for the S-Function block, allowing the callback methods to set and access the block properties during simulation or model updating.

About Run-Time Objects

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

Run-time objects do not support MATLAB sparse matrices. For example, if the variable `block` is a run-time object, the following line in a Level-2 M-file S-function produces an error:

```
block.Outport(1).Data = speye(10);
```

where the `speye` command forms a sparse identity matrix.

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

Level-2 M-File S-Function Template

Use the basic Level-2 M-file S-function template

```
matlabroot/toolbox/simulink/blocks/msfuntmpl_basic.m
```

to get a head start on creating a new Level-2 M-file S-function. The template contains skeleton implementation of the required callback methods defined by the Level-2 M-File S-function API. To write a more complicated S-function, use the annotated template

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

To create an M-file S-function, make a copy of the template and edit the copy as necessary to reflect the desired behavior of the S-function you are creating. The following two sections describe the contents of the M-file template. The section “Example of Writing a Level-2 M-File S-Function” on page 3-8 describes how to write a Level-2 M-file S-function that models a unit delay.

Level-2 M-File S-Function Callback Methods

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

A Level-2 M-file S-function must include the following callback methods:

- A `setup` function to initialize the basic S-function characteristics
- An `Outputs` function to calculate the S-function outputs

Your S-function can contain other methods, depending on the requirements of the block that the S-function defines. The methods defined by the Level-2 M-file S-function API generally correspond to similarly named methods defined by the C MEX-file S-function API. For information on when these methods are called during simulation, see “Process View” on page 4-77 in “How the Simulink Engine Interacts with C S-Functions” on page 4-77. For instructions on how to implement each callback method, see Chapter 9, “S-Function Callback Methods — Alphabetical List”.

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

Level-2 M-File Method	Equivalent C MEX-File Method
`setup` (see "Using the `setup` Method" on page 3-8)	`mdlInitializeSizes`
`CheckParameters`	`mdlCheckParameters`
`Derivatives`	`mdlDerivatives`
`Disable`	`mdlDisable`
`Enable`	`mdlEnable`
`InitializeCondition`	`mdlInitializeConditions`
`Outputs`	`mdlOutputs`
`PostPropagationSetup`	`mdlSetWorkWidths`
`ProcessParameters`	`mdlProcessParameters`
`Projection`	`mdlProjection`
`SetInputPortComplexSignal`	`mdlSetInputPortComplexSignal`
`SetInputPortDataType`	`mdlSetInputPortDataType`
`SetInputPortDimensions`	`mdlSetInputPortDimensionInfo`
SetInputPortDimensionsModeFcn	`mdlSetInputPortDimensionsModeFcn`
`SetInputPortSampleTime`	`mdlSetInputPortSampleTime`
`SetInputPortSamplingMode`	`mdlSetInputPortFrameData`
`SetOutputPortComplexSignal`	`mdlSetOutputPortComplexSignal`
`SetOutputPortDataType`	`mdlSetOutputPortDataType`
`SetOutputPortDimensions`	`mdlSetOutputPortDimensionInfo`
`SetOutputPortSampleTime`	`mdlSetOutputPortSampleTime`
`SimStatusChange`	`mdlSimStatusChange`
`Start`	`mdlStart`
`Terminate`	`mdlTerminate`
`Update`	`mdlUpdate`
`WriteRTW`	`mdlRTW`

Using the `setup` Method

The body of the `setup` method in a Level-2 M-file S-function initializes the instance of the corresponding Level-2 M-File S-Function block. In this respect, the `setup` method is similar to the `mdlInitializeSizes` and `mdlInitializeSampleTimes` callback methods implemented by C MEX S-functions. The `setup` method performs the following tasks:

- Initializing the number of input and output ports of the block.
- Setting attributes such as dimensions, data types, complexity, and sample times for these ports.
- Specifying the block sample time. See "How to Specify the Sample Time" in *Using Simulink* for more information on how to specify valid sample times.
- Setting the number of S-function dialog parameters.
- Registering S-function callback methods by passing the handles of local functions in the M-file S-function to the `RegBlockMethod` method of the S-Function block's run-time object. See the documentation for `Simulink.MSFcnRunTimeBlock` for information on using the `RegBlockMethod` method.

Example of Writing a Level-2 M-File S-Function

The following steps illustrate how to write a simple Level-2 M-file S-function. When applicable, the steps include examples from the S-function demo `*matlabroot*/toolbox/simulink/blocks/msfcn_unit_delay.m` used in the model `msfcndemo_sfundsc2.mdl`. All lines of code use the variable name `block` for the S-function run-time object.

1. Copy the Level-2 M-file S-function template `msfuntmpl_basic.m` to your working directory. If you change the file name when you copy the file, change the function name in the `function` line to the same name.

2. Modify the `setup` method to initialize the S-function's attributes. For this example:

   - Set the run-time object's `NumInputPorts` and `NumOutputPorts` properties to `1` to initialize one input port and one output port.
   - Invoke the run-time object's `SetPreCompInpPortInfoToDynamic` and `SetPreCompOutPortInfoToDynamic` methods to indicate that the

input and output ports inherit their compiled properties (dimensions, data type, complexity, and sampling mode) from the model.

- Set the `DirectFeedthrough` property of the run-time object's `InputPort` to `false` to indicate the input port does not have direct feedthrough. Retain the default values for all other input and output port properties that are set in your copy of the template file. The values set for the `Dimensions`, `DatatypeID`, and `Complexity` properties override the values inherited using the `SetPreCompInpPortInfoToDynamic` and `SetPreCompOutPortInfoToDynamic` methods.
- Set the run-time object's `NumDialogPrms` property to `1` to initialize one S-function dialog parameter.
- Specify that the S-function has an inherited sample time by setting the value of the runtime object's `SampleTimes` property to `[-1 0]`.
- Call the run-time object's `RegBlockMethod` method to register the following four callback methods used in this S-function.
  - `PostPropagationSetup`
  - `InitializeConditions`
  - `Outputs`
  - `Update`

  Remove any other registered callback methods from your copy of the template file. In the calls to `RegBlockMethod`, the first input argument is the name of the S-function API method and the second input argument is the function handle to the associated local function in the M-file S-function.

The following `setup` method from `msfcn_unit_delay.m` performs the previous list of steps:

```
function setup(block)

%% Register a single dialog parameter
block.NumDialogPrms  = 1;

%% Register number of input and output ports
block.NumInputPorts  = 1;
block.NumOutputPorts = 1;
```

```
%% Setup functional port properties to dynamically
%% inherited.
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

%% Hard-code certain port properties
block.InputPort(1).Dimensions        = 1;
block.InputPort(1).DirectFeedthrough = false;

block.OutputPort(1).Dimensions       = 1;

%% Set block sample time to inherited
block.SampleTimes = [-1 0];

%% Register methods
block.RegBlockMethod('PostPropagationSetup',@DoPostPropSetup);
block.RegBlockMethod('InitializeConditions',@InitConditions);
block.RegBlockMethod('Outputs',             @Output);
block.RegBlockMethod('Update',              @Update);
```

If your S-function needs continuous states, initialize the number of continuous states in the `setup` method using the run-time object's `NumContStates` property. Do not initialize discrete states in the `setup` method.

3 Initialize the discrete states in the `PostPropagationSetup` method. A Level-2 M-file S-function stores discrete state information in a DWork vector. The default `PostPropagationSetup` method in the template file suffices for this example.

The following `PostPropagationSetup` method from `msfcn_unit_delay.m`, named `DoPostPropSetup`, initializes one DWork vector with the name `x0`.

```
function DoPostPropSetup(block)

  %% Setup Dwork
  block.NumDworks = 1;
  block.Dwork(1).Name = 'x0';
  block.Dwork(1).Dimensions      = 1;
  block.Dwork(1).DatatypeID      = 0;
  block.Dwork(1).Complexity      = 'Real';
```

```
    block.Dwork(1).UsedAsDiscState = true;
```

If your S-function uses additional DWork vectors, initialize them in the `PostPropagationSetup` method, as well (see “Using DWork Vectors in Level-2 M-File S-Functions” on page 7-12).

4 Initialize the values of discrete and continuous states or other DWork vectors in the `InitializeConditions` or `Start` callback methods. Use the `Start` callback method for value that are initialized once at the beginning of the simulation. Use the `InitializeConditions` method for values that need to be reinitialized whenever an enabled subsystem containing the S-function is reenabled.

For this example, use the `InitializeConditions` method to set the discrete state’s initial condition to the value of the S-function’s dialog parameter. For example, the `InitializeConditions` method in `msfcn_unit_delay.m` is:

```
function InitConditions(block)

  %% Initialize Dwork
  block.Dwork(1).Data = block.DialogPrm(1).Data;
```

For S-functions with continuous states, use the `ContStates` run-time object method to initialize the continuous state date. For example:

```
 block.ContStates.Data(1) = 1.0;
```

5 Calculate the S-function’s outputs in the `Outputs` callback method. For this example, set the output to the current value of the discrete state stored in the DWork vector.

The `Outputs` method in `msfcn_unit_delay.m` is:

```
function Output(block)

  block.OutputPort(1).Data = block.Dwork(1).Data;
```

6 For an S-function with continuous states, calculate the state derivatives in the `Derivatives` callback method. Run-time objects store derivative data in their `Derivatives` property. For example, the following line sets the first state derivative equal to the value of the first input signal.

```
block.Derivatives(1).Data = block.InputPort(1).Data;
```

This example does not use continuous states and, therefore, does not implement the `Derivatives` callback method.

7. Update any discrete states in the `Update` callback method. For this example, set the value of the discrete state to the current value of the first input signal.

   The `Update` method in `msfcn_unit_delay.m` is:

   ```
   function Update(block)

     block.Dwork(1).Data = block.InputPort(1).Data;
   ```

8. Perform any cleanup, such as clearing variables or memory, in the `Terminate` method. Unlike C MEX S-functions, Level-2 M-file S-function are not required to have a `Terminate` method.

For information on additional callback methods, see "Level-2 M-File S-Function Callback Methods" on page 3-6. For a list of run-time object properties, see the reference page for `Simulink.MSFcnRunTimeBlock` and the parent class `Simulink.RunTimeBlock`.

Instantiating a Level-2 M-File S-Function

To use a Level-2 M-file S-function in a model, copy an instance of the Level-2 M-File S-Function block into the model. Open the Block Parameters dialog box for the block and enter the name of the M-file that implements your S-function into the **M-file name** field. If your S-function uses any additional parameters, enter the parameter values as a comma-separated list in the Block Parameters dialog box **Parameters** field.

Operations for Variable-Size Signals

Following are modifications to the Level-2 M-File S-functions template (`msfuntmpl_basic.m`) and additional operations that allow you to use variable-size signals.

```
function setup(block)
% Register the properties of the output port
```

```
block.OutputPort(1).DimensionsMode = 'Variable';
block.RegBlockMethod('SetInputPortDimensionsMode',  @SetInputDimsMode);

function DoPostPropSetup(block)
%Register dependency rules to update current output size of output port a
block.AddOutputDimsDependencyRules(a, [b c], @setOutputVarDims);

%Confi gure output port b to have the same dimensions as input port a
block.InputPortSameDimsAsOutputPort(a,b);

%Configure DWork a to have its size reset when input size changes.
block.DWorkRequireResetForSignalSize(a,true);

function SetInputDimsMode(block, port, dm)
% Set dimension mode
block.InputPort(port).DimensionsMode = dm;
block.OutputPort(port).DimensionsMode = dm;

function setOutputVarDims(block, opIdx, inputIdx)
% Set current (run-time) dimensions of the output
outDimsAfterReset = block.InputPort(inputIdx(1)).CurrentDimensions;
block.OutputPort(opIdx).CurrentDimensions = outDimsAfterReset;
```

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

Generating code for a model containing a Level-2 M-file S-function requires that you provide a corresponding Target Language Compiler (TLC) file. You do not need a TLC file to accelerate a model containing a Level-2 M-file S-function. The Simulink Accelerator software runs Level-2 M-file S-functions in interpreted mode. See "Inlining M-File S-Functions" in the *Real-Time Workshop User's Guide* for more information on writing TLC files for M-file S-functions.

M-File S-Function Demos

The Level-2 M-file S-function demos provide a set of self-documenting models that illustrate the use of Level-2 M-file S-functions. Enter `sfundemos` at the MATLAB command prompt to view the demos.

Maintaining Level-1 M-File S-Functions

In this section...

About the Maintenance of Level-1 M-File S-Functions

Note The information provided in this section is intended only for use in maintaining existing Level-1 M-file S-functions. Use the more capable Level-2 API to develop new M-file S-functions (see "Writing Level-2 M-File S-Functions" on page 3-4). Level-1 M-file S-functions support a much smaller subset of the S-function API then Level-2 M-file S-functions, and their features are limited compared to built-in blocks.

A Level-1 M-file S-function is 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, the Simulink engine repeatedly invokes *f*, using the `flag` argument to indicate the task (or tasks) to be performed for a particular invocation. The S-function performs the task and returns the results in an output vector.

A template implementation of a Level-1 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.

Level-1 M-File S-Function Arguments

The Simulink engine passes the following arguments to a Level-1 M-file 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 Level-2 M-file S-function method for each value.

Flag Argument

Level-1 Flag	Level-2 Callback Method	Description
0	`setup`	Defines basic S-Function block characteristics, including sample times, initial conditions of continuous and discrete states, and the `sizes` array (see "Defining S-Function Block Characteristics" on page 3-17 for a description of 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.

Flag Argument (Continued)

Level-1 Flag	Level-2 Callback Method	Description
4	`mdlOutputs` method updates the run-time object `NextTimeHit` property	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 the `setup` method.
9	`mdlTerminate`	Performs any necessary end-of-simulation tasks.

Level-1 M-File S-Function Outputs

A Level-1 M-file S-function 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. Level-1 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 "How to Specify the Sample Time" in *Using Simulink* for information on how to specify a sample times and offsets).

  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 the Simulink engine to execute the S-function at the following times: `[0 0.1 0.25 0.5 0.75 1 1.1 ...]`. Your S-function must decide at every sample time which task to perform at that sample time.

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

Defining S-Function Block Characteristics

For the Simulink engine to recognize a Level-1 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 provide this information, call the `simsizes` function at the beginning of the S-function.

```
sizes = simsizes;
```

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

Fields in the sizes Structure

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

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

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

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

Processing S-Function Parameters

When invoking a Level-1 M-file S-function, the Simulink engine always passes the standard block parameters, `t`, `x`, `u`, and `flag`, to the S-function as function arguments. The engine 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 Block Parameters dialog box (see “Passing Parameters to S-Functions” on page 1-5). If the block dialog specifies additional parameters, the engine 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 *`matlabroot`*`/toolbox/simulink/blocks` directory for an example of an S-function that uses block-specific parameters.

Converting Level-1 M-File S-Functions to Level-2

You can convert Level-1 M-file S-functions to Level-2 M-file S-functions by mapping the code associated with each Level-1 S-function flag to the appropriate Level-2 S-function callback method. See the Flag Arguments table for a mapping of Level-1 flags to Level-2 callback methods. In addition:

- Store discrete state information for Level-2 M-file S-functions in DWork vectors, initialized in the `PostPropagationSetup` method.
- Access Level-2 M-file S-function dialog parameters using the `DialogPrm` run-time object property, instead of passing them into the S-function as function arguments.
- For S-functions with variable sample times, update the `NextTimeHit` run-time object property in the `Outputs` method to set the next sample time hit for the Level-2 M-file S-function.

For example, the following table shows how to convert the Level-1 M-file S-function `sfundsc2.m` to a Level-2 M-file S-function. The example uses the Level-2 M-file S-function template `msfuntmpl_basic.m` as a starting point when converting the Level-1 M-file S-function. The line numbers in the table corresponds to the lines of code in `sfundsc2.m`.

Line #	Code in sfundsc2.m	Code in Level-2 M-file (sfundsc2_level2.m)

1	`function [sys,x0,str,ts]= ...`
	`  sfundsc2(t,x,u,flag)`
		`function sfundsc2(block)`
		`  setup(block);`
		
		The syntax for the `function` line changes to accept one input argument `block`, which is the Level-2 M-File S-Function block's run-time object. The main body of the Level-2 M-file S-function contains a single line that calls the local `setup` function.

13 - 19	`switch flag,`
	
	`case 0,`
	`[sys,x0,str,ts] = ...`
	`   mdlInitializeSizes;`
		`function setup(block)`
		
		The `flag` value of zero corresponds to calling the `setup` method. A Level-2 M-file S-function does not use a `switch` statement to invoke the callback methods. Instead, the local `setup` function registers callback methods that are directly called during simulation.

24 - 31	`case 2,`
	`   sys = mdlUpdate(t,x,u);`
	
	`case 3,`
	`   sys = mdlOutputs(t,x,u);`
		The `setup` function registers two local functions associated with `flag` values of `2` and `3`.
		
		`block.RegBlockMethod('Outputs' ,@Output);`
		`block.RegBlockMethod('Update'  ,@Update);`

Line #	Code in sfundsc2.m	Code in Level-2 M-file (sfundsc2_level2.m)

53 - 66	`sizes = simsizes;`
	
	`sizes.NumContStates  = 0;`
	`sizes.NumDiscStates  = 1;`
	`sizes.NumOutputs     = 1;`
	`sizes.NumInputs      = 1;`
	`sizes.DirFeedthrough = 0;`
	`sizes.NumSampleTimes = 1;`
	
	`sys = simsizes(sizes);`
	
	`x0  = 0;`
	`str = [];`
	`ts  = [.1 0];`
		The `setup` function also initializes the attributes of the Level-2 M-file S-function:
		
		`block.NumInputPorts  = 1;`
		`block.NumOutputPorts = 1;`
		`block.InputPort(1).Dimensions        = 1;`
		`block.InputPort(1).DirectFeedthrough = false;`
		`block.OutputPort(1).Dimensions       = 1;`
		`block.NumDialogPrms     = 0;`
		`block.SampleTimes = [0.1 0];`
		
		Because this S-function has discrete states, the `setup` method registers the `PostPropagationSetup` callback method to initialize a DWork vector and the `InitializeConditions` callback method to set the initial state value.
		
		`block.RegBlockMethod('PostPropagationSetup',...`
		` @DoPostPropSetup);`
		`block.RegBlockMethod('InitializeConditions', ...`
		` @InitConditions);`

56	`sizes.NumDiscStates  = 1;`	The `PostPropagationSetup` method initializes the DWork vector that stores the single discrete state.
		
		`function DoPostPropSetup(block)`
		
		`  %% Setup Dwork`
		`  block.NumDworks = 1;`
		`  block.Dwork(1).Name = 'x0';`
		`  block.Dwork(1).Dimensions      = 1;`
		`  block.Dwork(1).DatatypeID      = 0;`
		`  block.Dwork(1).Complexity      = 'Real';`
		`  block.Dwork(1).UsedAsDiscState = true;`

Line #	Code in sfundsc2.m	Code in Level-2 M-file (sfundsc2_level2.m)

64	`x0  = 0;`	The `InitializeConditions` method initializes the discrete state value.
		
		`function InitConditions(block)`
		
		`%% Initialize Dwork`
		`block.Dwork(1).Data = 0`

77 - 78	`function sys = ...`
	`   mdlUpdate(t,x,u)`
	
	`sys = u;`
		The `Update` method calculates the next value of the discrete state.
		
		`function Update(block)`
		`block.Dwork(1).Data = block.InputPort(1).Data;`

88 - 89	`function sys = ...`
	`   mdlOutputs(t,x,u)`
	`sys = x;`
		The `Outputs` method calculates the S-function's output.
		
		`function Output(block)`
		`block.OutputPort(1).Data = block.Dwork(1).Data;`

Writing S-Functions in C

Introduction

In this section...

About Writing C S-Functions

A C MEX S-function must provide information about the function to the Simulink engine during the simulation. As the simulation proceeds, the engine, the ODE solver, and the C MEX S-function 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, the Simulink engine interacts with a C MEX 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. However, the S-function is free to perform the task in each method according to the functionality the S-function implements. For example, the `mdlOutputs` method must compute the block outputs at the current simulation time. However, the S-function can calculate these outputs in any way that is appropriate for the function. This callback-based API allows you to create S-functions, and hence custom blocks, of any desired functionality.

The set of callback methods that C MEX S-functions can implement is larger than that available for M-file S-functions. See Chapter 9, "S-Function Callback Methods — Alphabetical List" for descriptions of the callback methods that a C MEX S-function can implement. C MEX S-functions are required to implement only a small subset of the callback methods in the S-function API. If your block does not implement a particular feature, such as matrix signals, you are free to omit the callback methods needed 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 the Simulink engine calls when interacting with the S-function. The engine subsequently invokes other S-function methods (all starting with `mdl`). At the end of a simulation, the engine calls `mdlTerminate`.

Creating C MEX S-Functions

You can create C MEX S-functions using any of the following approaches:

- Handwritten S-function — You can write a C MEX S-function from scratch. ("Example of a Basic C MEX S-Function" on page 4-43 provides a step-by-step example.) See "Templates for C S-Functions" on page 4-50 for a complete skeleton implementation of a C MEX S-function that you can use as a starting point for creating your own S-functions.
- S-Function Builder — This block builds a C MEX S-function from specifications and code fragments that you supply using a graphical user interface. This eliminates the need for you to write S-functions from scratch. See "Building S-Functions Automatically" on page 4-5 for more information about the S-Function Builder.

- Legacy Code Tool — This utility builds a C MEX S-function from existing C code and specifications that you supply using MATLAB M-code. See “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” on page 4-55 for more information about integrating legacy C code into Simulink models.

Each of these approaches involves a tradeoff between the ease of writing an S-function and the features supported by the S-function. Although handwritten S-functions support the widest range of features, they can be difficult to write. The S-Function Builder block simplifies the task of writing C MEX S-functions but supports fewer features. The Legacy Code Tool provides the easiest approach to creating C MEX S-functions from existing C code but supports the fewest features. See Chapter 2, “Selecting an S-Function Implementation” for more information on the features and limitations of each of these approaches to writing a C MEX S-function.

In addition to these three approaches, the Real-Time Workshop product provides a method for generating a C MEX S-function from a graphical subsystem. If you are new to writing C MEX S-functions, you can build portions of your application in a Simulink subsystem and use the S-function target to convert it to an S-function. The generated files provides insight on how particular blocks can be implemented within an S-function. See “Creating Component Object Libraries and Enhancing Simulation Performance” in the *Real-Time Workshop User’s Guide* for details and limitations on using the S-function target.

Building S-Functions Automatically

In this section...

About Building S-Functions Automatically

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

Note For examples of using the S-Function Builder to build S-functions, see the C-file S-functions subsystem of the S-function demos provided with the Simulink product. To display the demos, enter `sfundemos` at the MATLAB command line (see "S-Function Examples" on page 1-21 for more information).

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

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

   Note This directory must be on the MATLAB path.

   ---

2. If you wish to connect a bus signal to the Input or Output port of the S-Function Builder, you must first create a bus object. You perform this task interactively using the Simulink Bus Editor. (For more information, see "Using the Bus Editor". Alternatively, you can use `Simulink.Bus` as follows.

   a. At the MATLAB command line, enter:

```
a = Simulink.Bus
```

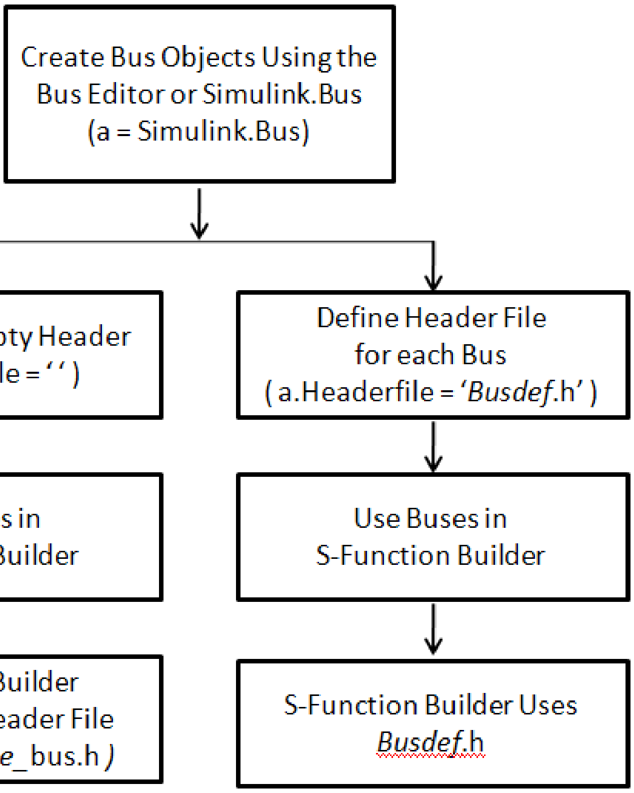
As a result, the `HeaderFile` for the bus defaults to the empty string:

```
a =

Simulink.Bus
    Description: ''
     HeaderFile: ''
       Elements: [0x1 double]
```

b If you wish to specify the header file for the bus, then at the MATLAB command line:

```
a.Headerfile = 'Busdef.h'
```

If you do not specify a header file, Simulink automatically generates *`Sfunctionname`*`_bus.h`

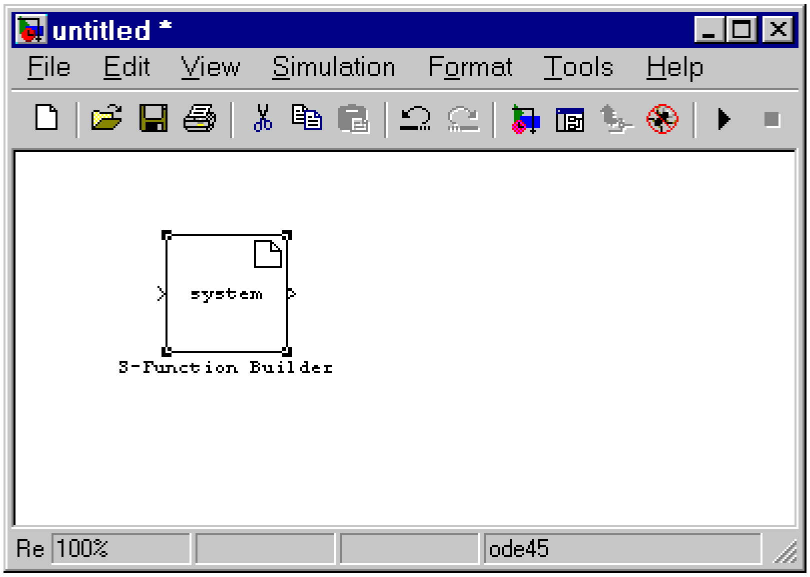

Create Bus Objects Using the Bus Editor or Simulink.Bus (a = Simulink.Bus)
Use Default Empty Header (a.Headerfile = ‘ ’)
Define Header File for each Bus (a.Headerfile = ‘Busdef.h’)
Use Buses in S-Function Builder
Use Buses in S-Function Builder
S-Function Builder Generates a Header File (Sfunctionname_bus.h)
S-Function Builder Uses Busdef.h

For a demonstration on how to use the S-Function Builder with a bus, see the `S-Function Builder with buses` example by entering the following command at the MATLAB command line:

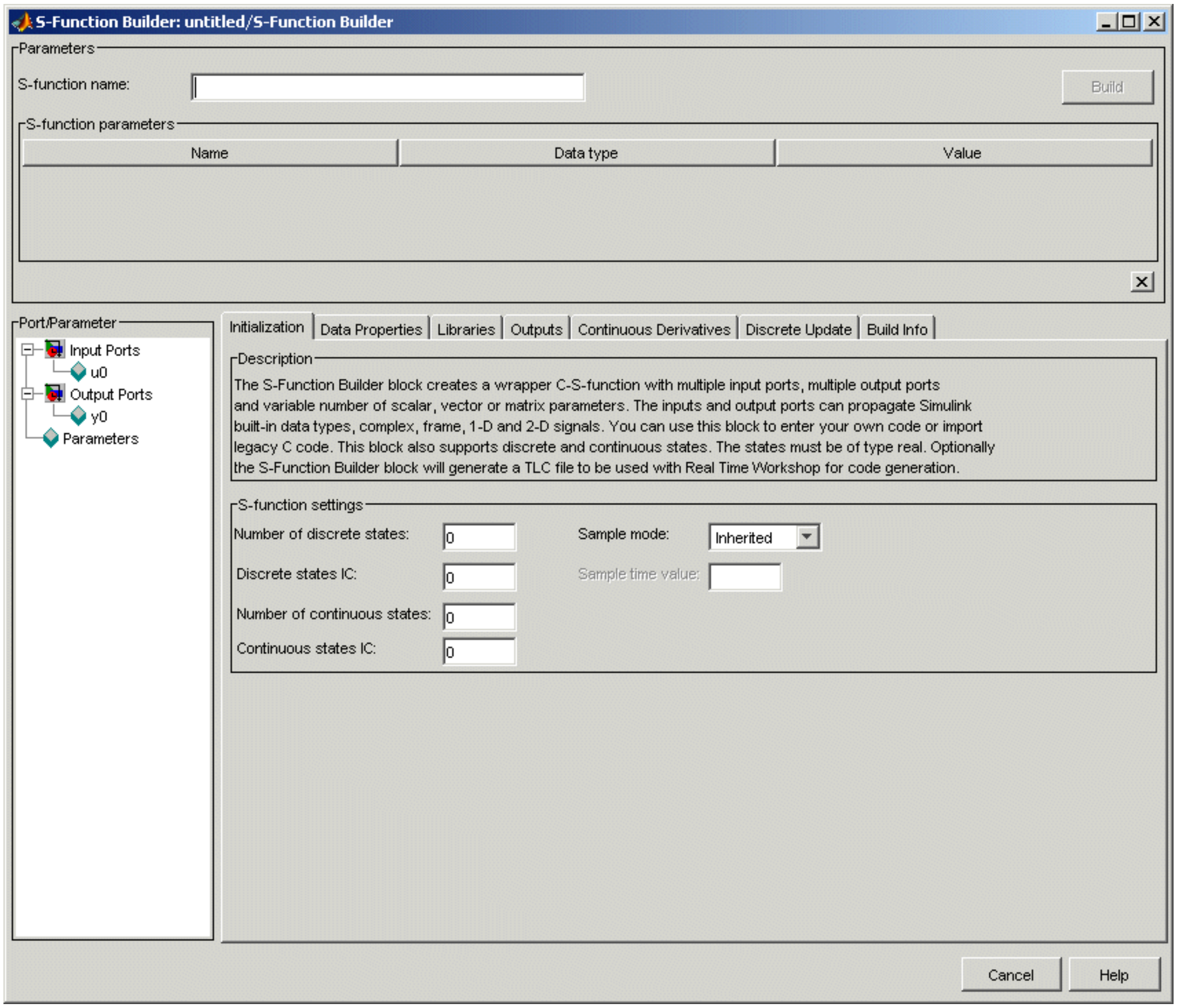
```
open_system([matlabroot,'/toolbox/simulink/simdemos/
simfeatures/sfbuilder_bususage.mdl']);
```

3 Create a new Simulink model.

4 Copy an instance of the S-Function Builder block from the User-Defined Functions library in the Library Browser into the new model.



5 Double-click the block to open the S-Function Builder dialog box (see “S-Function Builder Dialog Box” on page 4-12).



The S-Function Builder block creates a wrapper C-S-function with multiple input ports, multiple output ports and variable number of scalar, vector or matrix parameters. The inputs and output ports can propagate Simulink built-in data types, complex, frame, 1-D and 2-D signals. You can use this block to enter your own code or import legacy C code. This block also supports discrete and continuous states. The states must be of type real. Optionally the S-Function Builder block will generate a TLC file to be used with Real Time Workshop for code generation.

Number of discrete states: 0
Discrete states IC: 0
Number of continuous states: 0
Continuous states IC: 0
Sample mode: Inherited
Sample time value:

6 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 4-12).

7 If you have not already done so, configure the `mex` command to work on your system.

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

8 Click **Build** on the S-Function Builder to start the build process.

The S-Function Builder 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 4-11).

9 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.

Alternatively, you can deploy the generated S-function without using the S-Function Builder block or exposing the underlying C source file. To do this:

1 Open the Simulink model that will include the S-function.

2 Copy an S-Function block from the User-Defined Functions library in the Library Browser into the model.

3 Double-click on the S-Function block.

4 In the Block Parameters dialog box that opens, enter the name of the executable file generated by the S-Function Builder into the **S-function name** edit field.

5 Enter any parameters needed by the S-function into the **S-function parameters** edit field. Enter the parameters in the order they appear in the S-Function Builder dialog box.

6 Click **OK** on the S-Function Block Parameters dialog box.

You can use the generated executable file, for example, the `.mexw32` file, in any S-Function block in any model as long as the executable file is on the MATLAB path.

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 specify in the **S-function name** field of the S-Function Builder 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 the generated S-function to run in Simulink Rapid Accelerator mode and allows for inlining the S-function during code generation. In addition, this file generates code for the S-function in Accelerator mode, thus allowing the model to run faster.

- `sfun_bus.h`

  If you specify any `Input port` or `Output port` as a bus in the Data Properties pane of the S-Function Builder dialog box, but do not specify a header file, then the S-Function Builder automatically generates this header file.

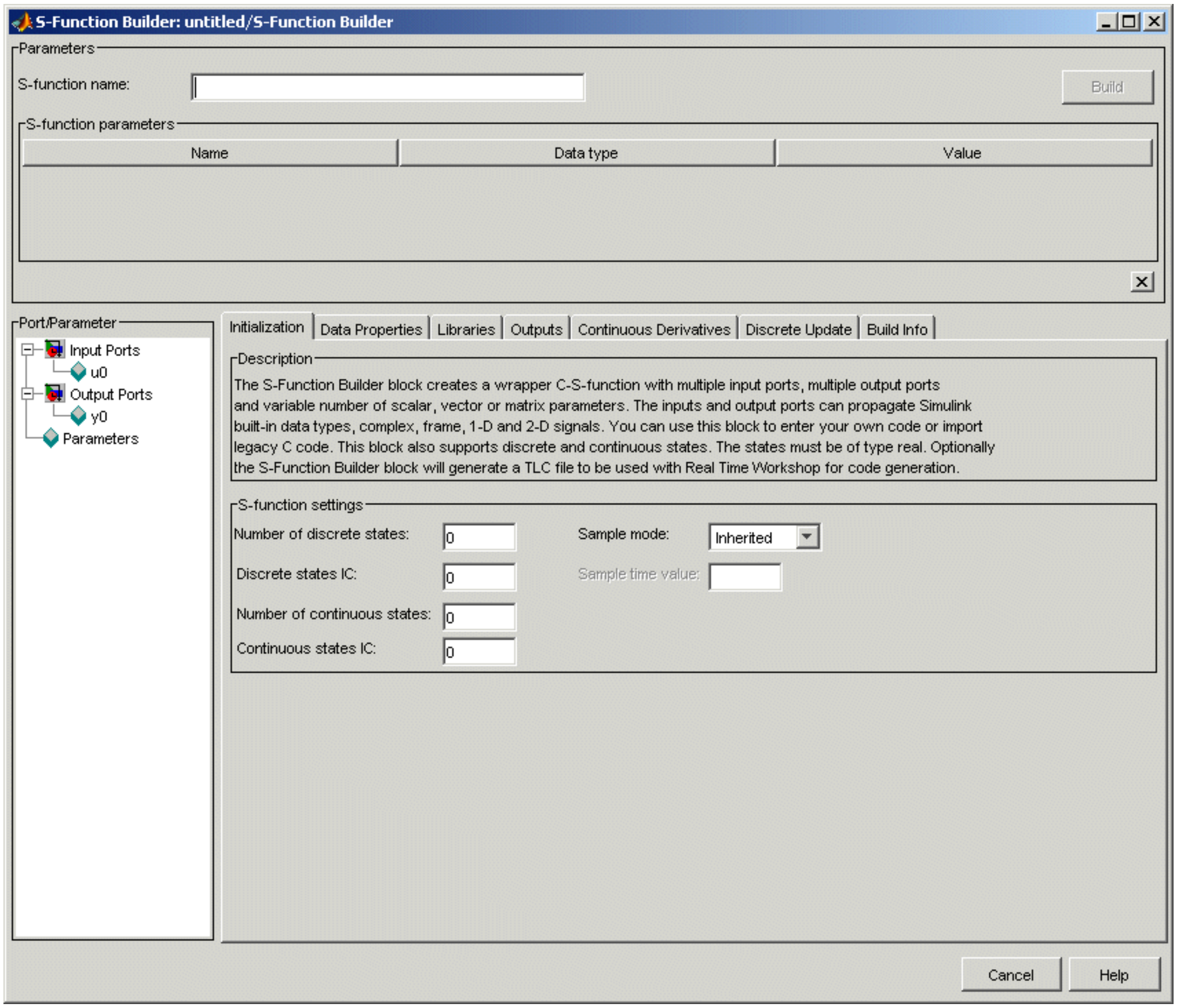
After generating the S-function source code, the S-Function Builder uses the `mex` command to build the MEX-file representation of the S-function from the generated source code and any external custom source code and libraries that you specified.

S-Function Builder Dialog Box

In this section...

About S-Function Builder

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



S-function name:

The S-Function Builder block creates a wrapper C-S-function with multiple input ports, multiple output ports and variable number of scalar, vector or matrix parameters. The inputs and output ports can propagate Simulink built-in data types, complex, frame, 1-D and 2-D signals. You can use this block to enter your own code or import legacy C code. This block also supports discrete and continuous states. The states must be of type real. Optionally the S-Function Builder block will generate a TLC file to be used with Real Time Workshop for code generation.

Number of discrete states:	0	Sample mode:	Inherited
Discrete states IC:	0	Sample time value:	
Number of continuous states:	0		
Continuous states IC:	0		

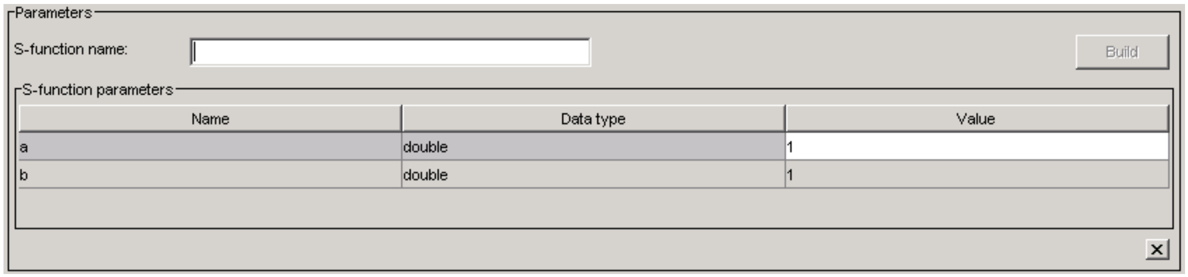
The dialog box contains controls that let you enter information needed for the S-Function Builder block to build an S-function to your specifications. The controls are grouped into panes. See the following sections for information on the panes and the controls that they contain.

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

See “Example: Modeling a Two-Input/Two-Output System” on page 4-37 for an example showing how to use the S-Function Builder to model a two-input/two-output discrete state-space system.

Parameters/S-Function Name Pane

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



Name	Data type	Value
a	double	1
b	double	1

The pane contains the following controls.

S-function name

Specifies the name of the target S-function.

S-function parameters

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

- **Name** — Name of the parameter. Define and modify this property from the “Parameters Pane” on page 4-22.
- **Data type** — Lists the data type of the parameter. Define and modify this property from the “Parameters Pane” on page 4-22.

- **Value** — Specifies the value of the parameter. Enter a valid MATLAB expression in this field.

Build/Save

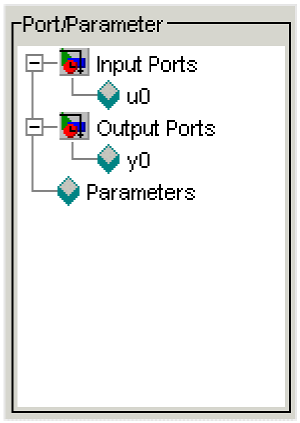
Use this button to generate the C source code and executable MEX-file from the information you entered in the S-Function Builder. If the button is labeled **Build**, the S-Function Builder generates the source code and executable MEX-file. If the button is labeled **Save**, it generates only the C source code. Use the **Save code only** check box on the **Build Info** pane to toggle the functionality of this button.

Hide/Show S-function editing tabs

Use the small button at the bottom-right of the **Parameters/S-Function Name** pane, to collapse and expand the bottom portion of the S-Function Builder dialog box.

Port/Parameter Pane

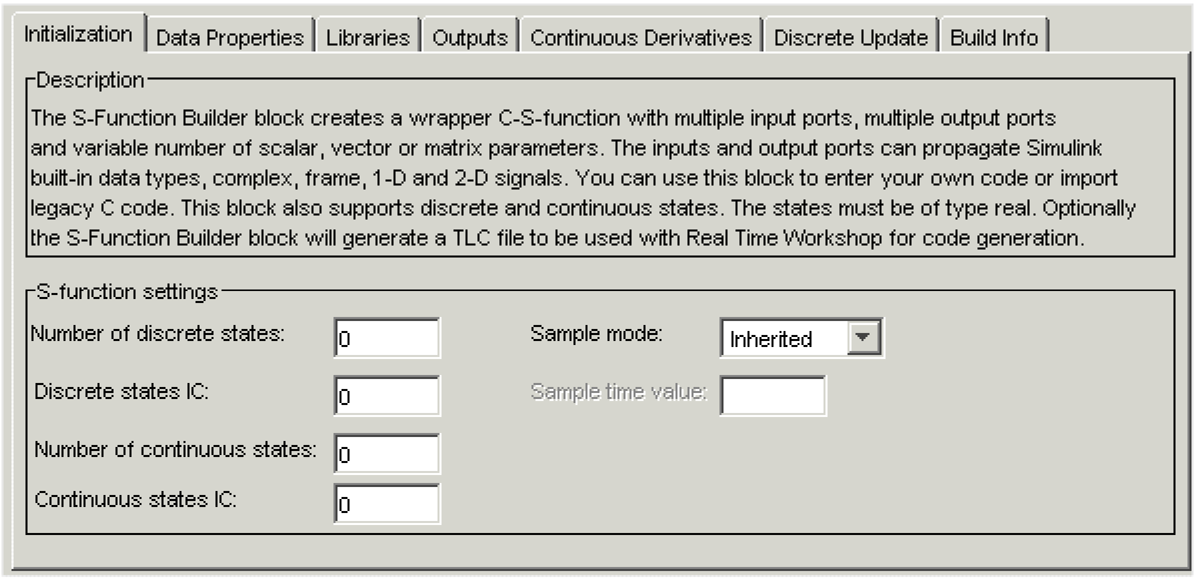
This pane displays the ports and parameters that the dialog box specifies for the target S-function.



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

Initialization Pane

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



The S-Function Builder block creates a wrapper C-S-function with multiple input ports, multiple output ports and variable number of scalar, vector or matrix parameters. The inputs and output ports can propagate Simulink built-in data types, complex, frame, 1-D and 2-D signals. You can use this block to enter your own code or import legacy C code. This block also supports discrete and continuous states. The states must be of type real. Optionally the S-Function Builder block will generate a TLC file to be used with Real Time Workshop for code generation.

Number of discrete states:	0
Discrete states IC:	0
Number of continuous states:	0
Continuous states IC:	0
Sample mode:	Inherited

The S-Function Builder uses the information that you enter on this pane to generate the `mdlInitializeSizes` callback method. The Simulink engine invokes this method during the model initialization phase of the simulation to obtain basic information about the S-function. (See "How the Simulink Engine Interacts with C S-Functions" on page 4-77 for more information on the model initialization phase.)

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

Number of discrete states

Number of discrete states in the S-function.

Discrete states IC

Initial conditions of the discrete states in the S-function. 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 in the S-function.

Continuous states IC

Initial conditions of the continuous states in the S-function. 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 **Sample time value** field of the S-Function Builder dialog box.

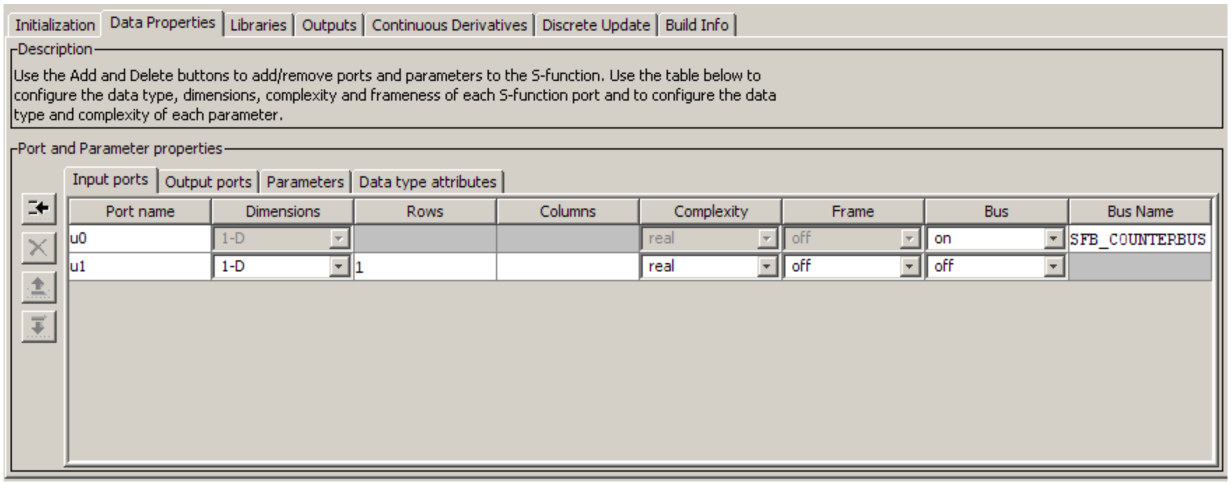
Sample time value

Scalar value indicating the interval between updates of the S-function outputs. This field is enabled only if you select `Discrete` as the **Sample mode**.

Note The S-Function Builder does not currently support multiple-block sample times or a nonzero offset time.

Data Properties Pane

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



Description

Use the Add and Delete buttons to add/remove ports and parameters to the S-function. Use the table below to configure the data type, dimensions, complexity and frameness of each S-function port and to configure the data type and complexity of each parameter.

Port and Parameter properties

Port name	Dimensions	Rows	Columns	Complexity	Frame	Bus	Bus Name
u0	1-D			real	off	on	SFB_COUNTERBUS
u1	1-D	1		real	off	off	

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

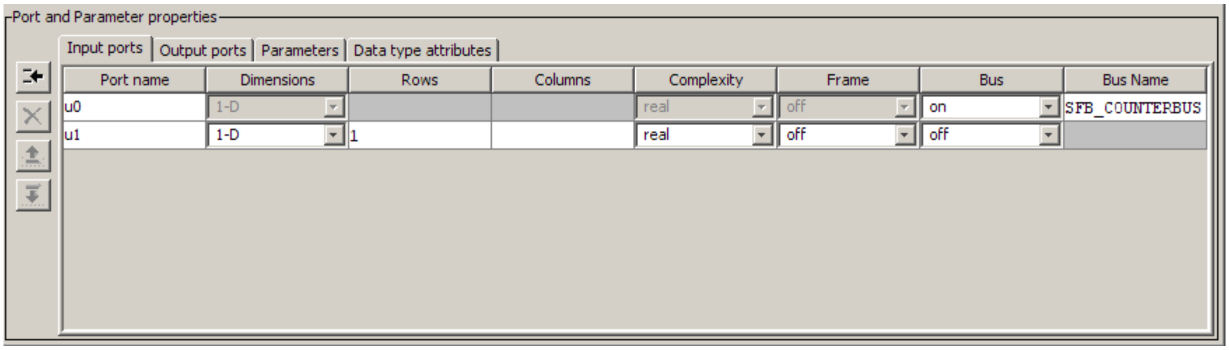
- To add a port or a parameter, click the **Add** button (the top button in the column of buttons).
- To delete the currently selected port or parameter, click the **Delete** button (located beneath the **Add** button).
- To move the currently selected port or parameter up one position in the corresponding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

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

- "Input Ports Pane" on page 4-19
- "Output Ports Pane" on page 4-21
- "Parameters Pane" on page 4-22
- "Data Type Attributes Pane" on page 4-23

Input Ports Pane

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



Port name	Dimensions	Rows	Columns	Complexity	Frame	Bus	Bus Name
u0	1-D			real	off	on	SFB_COUNTERBUS
u1	1-D	1		real	off	off	

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 Builder block. Each row of the table corresponds to a port. Each entry in the row displays a property of the port as follows.

Port name

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

Dimensions

Lists the number of dimensions of the input signal accepted by the port. To display a list of supported dimensions, click the adjacent button. To change the port dimensionality, select a new value from the list. Specify `1-D` to size the signal dynamically, regardless of the actual dimensionality of the signal.

Rows

Specifies the size of the first (or only) dimension of the input signal. Specify `-1` to size the signal dynamically.

Columns

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

Note For input signals with two dimensions, if the rows dimension is dynamically sized, the columns dimension must also be dynamically sized or set to 1. If the columns dimension is set to some other value, the S-function will compile, but any simulation containing this S-function will not run due to an invalid dimension specification.

Complexity

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

Frame

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

Bus

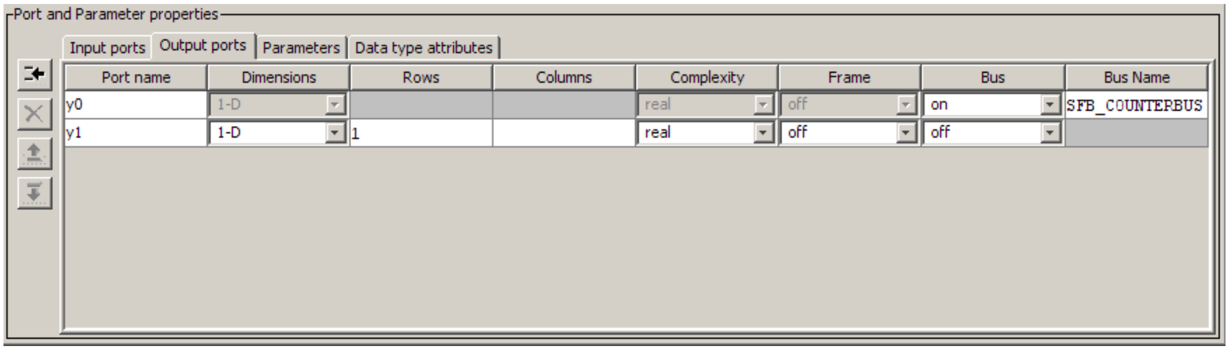
If the input signal to the S-Function Builder block is a bus, then use the drop-down menu in the `Bus` column to select '`on`'.

Bus Name

Step 2 of the “Building S-Functions Automatically” on page 4-5 instructs you to create a bus object, if your input signal is a bus. In the field provided in the Bus Name column, enter the bus name that you defined while creating the inport bus object.

Output Ports Pane

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



Port and Parameter properties

Input ports | Output ports | Parameters | Data type attributes

Port name	Dimensions	Rows	Columns	Complexity	Frame	Bus	Bus Name
y0	1-D			real	off	on	SFB_COUNTERBUS
y1	1-D	1		real	off	off	

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

Port name

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

Dimensions

Lists the number of dimensions of signals output by the port. To display a list of supported dimensions, click the adjacent button. To change the port dimensionality, select a new value from the list. Specify `1-D` to size the signal dynamically, regardless of the actual dimensionality of the signal.

Rows

Specifies the size of the first (or only) dimension of the output signal. Specify `-1` to size the signal dynamically.

Columns

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

Note For output signals with two dimensions, if one of the dimensions is dynamically sized the other dimension must also be dynamically sized or set to 1. If the second dimension is set to some other value, the S-function will compile, but any simulation containing this S-function will not run due to an invalid dimension specification. In some cases, the calculations that determine the dimensions of dynamically sized output ports may be insufficient and both dimensions of the 2-D output signal may need to be hard coded.

Complexity

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

Frame

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

Bus

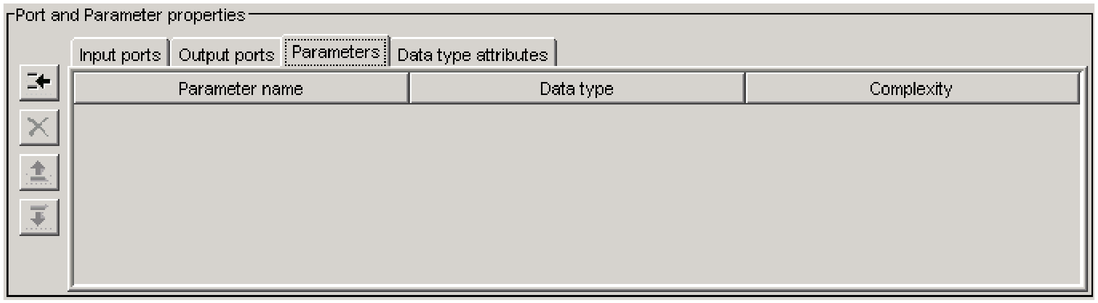
If the output signal to the S-Function Builder block is a bus, then use the drop-down menu in the `Bus` column to select '`on`'.

Bus Name

Step 2 of the "Building S-Functions Automatically" on page 4-5 instructs you to create a bus object. In the field provided in the `Bus Name` column, enter the name that you defined while creating the outport bus object.

Parameters Pane

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



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

Parameter name

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

Data type

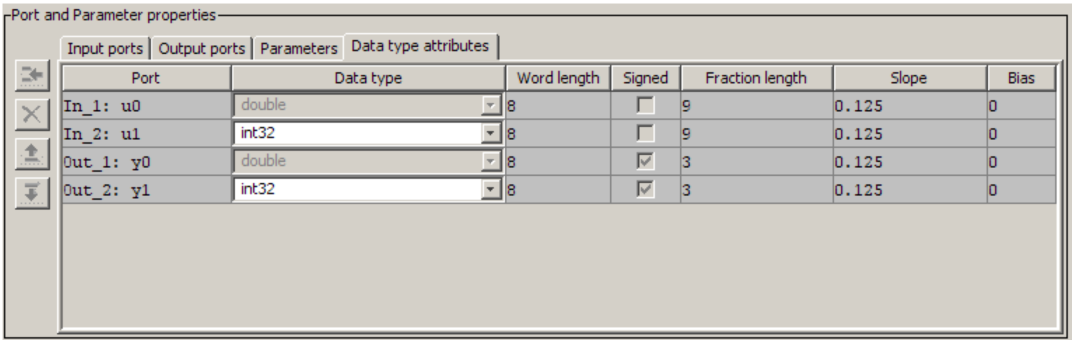
Lists the data type of the parameter. Click the adjacent button to display a list of supported data types. To change the parameter data type, select a new type from the list.

Complexity

Specifies whether the parameter has real or complex values.

Data Type Attributes Pane

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



Port	Data type	Word length	Signed	Fraction length	Slope	Bias
In_1: u0	double	8	☐	9	0.125	0
In_2: u1	int32	8	☐	9	0.125	0
Out_1: y0	double	8	☑	3	0.125	0
Out_2: y1	int32	8	☑	3	0.125	0

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

Port

Name of the port. This field displays the name entered in the **Input ports** and **Output ports** panes. You cannot edit this field.

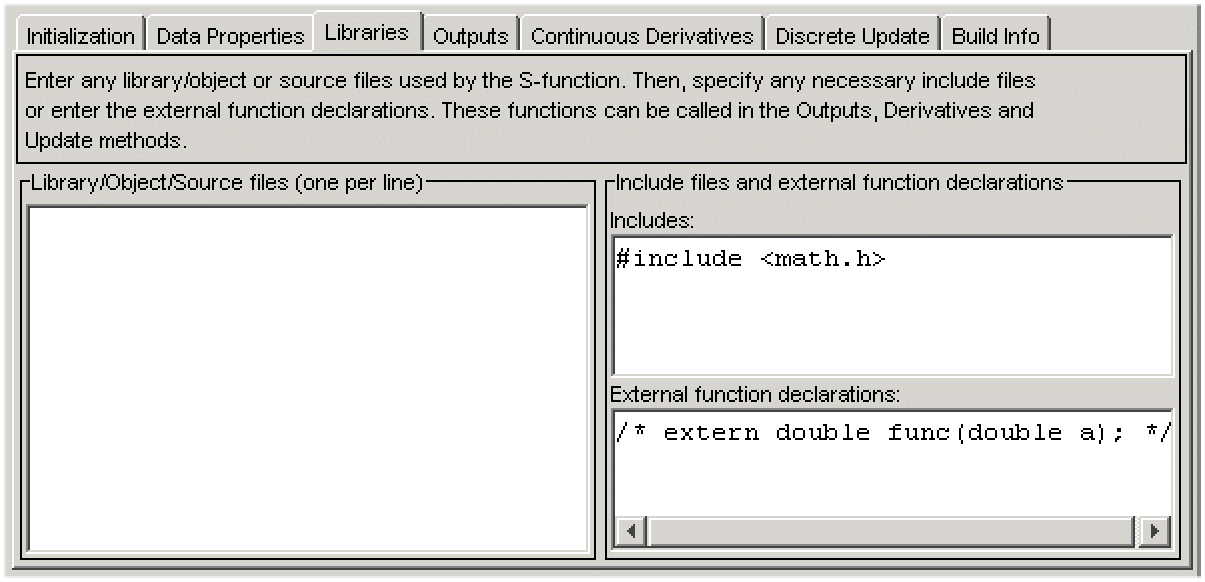
Data Type

Data type of the port. Click the adjacent button to display a list of supported data types. To change the data type, select a different data type from the list.

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

Libraries Pane

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



Enter any library/object or source files used by the S-function. Then, specify any necessary include files or enter the external function declarations. These functions can be called in the Outputs, Derivatives and Update methods.

Library/Object/Source files (one per line)

Include files and external function declarations

Includes:

```
#include <math.h>
```

External function declarations:

```
/* extern double func(double a); */
```

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 name. If the file resides in another directory, you must specify the full path of the file.

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

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

- `c:\customfolder\customfunctions.lib`

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

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

```
SRC_PATH d:\externalsource
LIB_PATH $MATLABROOT\customobjs
LIB_PATH c:\customfolder
customfunctions.lib
userfunctions.obj
freesource.c
```

As this example illustrates, you can use `LIB_PATH` to specify both object and library file paths. You can include the library name in the `LIB_PATH` declaration, however you must place the object file name on a separate line. The tag `$MATLABROOT` indicates a path relative to the MATLAB installation. You include multiple `LIB_PATH` entries on separate lines. The paths are searched in the order specified.

You can also enter preprocessor (`-D`) directives in this field, for example,

```
-DDEBUG
```

Each directive must reside on a separate line.

Note Do not put quotation marks around the library path, even if the path name has spaces in it. If you add quotation marks, the compiler will not find the library.

Includes

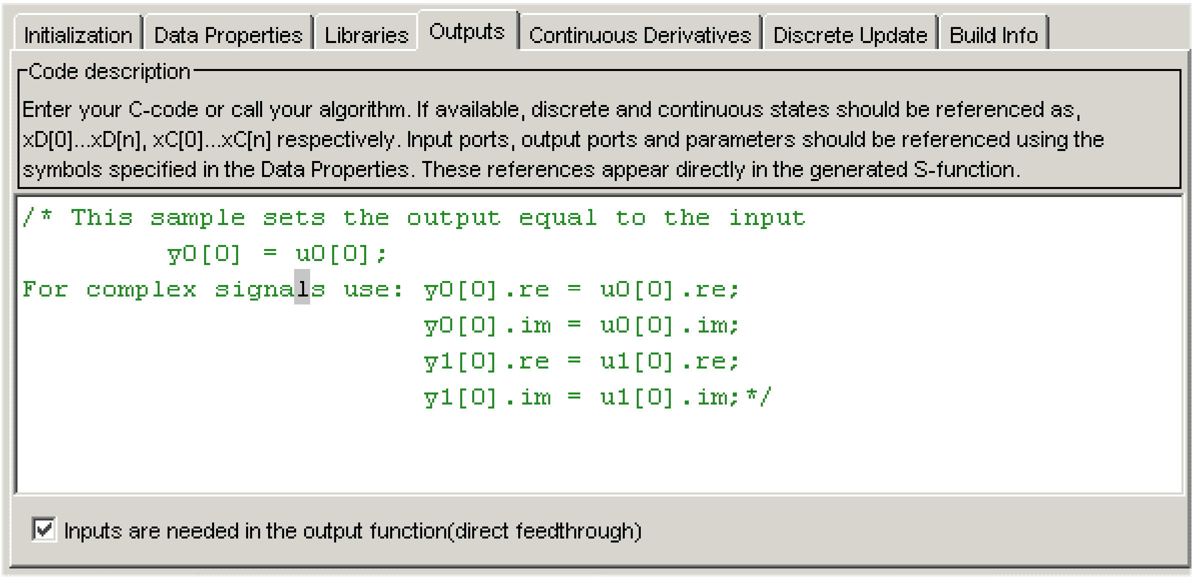
Header files containing declarations of functions, variables, and macros referenced by custom code that you enter elsewhere on the S-Function Builder dialog box. Specify each file on a separate line as `#include` statements. Use brackets to enclose the names of standard C header files (e.g., `#include <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 use the `INC_PATH` tag in the **Library/Object/Source files** field to set the include path for the S-Function Builder to the directories containing the include files (see "Library/Object/Source files" on page 4-25).

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 states or outputs 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.



Code description

Enter your C-code or call your algorithm. If available, discrete and continuous states should be referenced as, xD[0]...xD[n], xC[0]...xC[n] respectively. Input ports, output ports and parameters should be referenced using the symbols specified in the Data Properties. These references appear directly in the generated S-function.

```
/* This sample sets the output equal to the input
         y0[0] = u0[0];
For complex signals use: y0[0].re = u0[0].re;
                         y0[0].im = u0[0].im;
                         y1[0].re = u1[0].re;
                         y1[0].im = u1[0].im;*/
```

[✓] Inputs are needed in the output function(direct feedthrough)

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. The Simulink engine 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 output. The `mdlOutputs` method in turn invokes the wrapper function containing your output code. Your output code then actually computes and returns the S-function output.

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

Argument	Description
`u0, u1, ...  uN`	Pointers to arrays containing the inputs to the S-function, where `N` is the number of input ports specified on the **Input ports** pane found on the **Data Properties** pane. The names of the arguments that appear in the outputs wrapper function are the same as the names found on the **Input ports** pane. The width of each array is the same as the input width specified for each input on the **Input ports** pane. If you specified -1 as an input width, the width of the array is specified by the wrapper function's `u_width` argument (see below).
`y0, y1, ...  yN`	Pointer to arrays containing the outputs of the S-function, where `N` is the number of output ports specified on the **Output ports** pane found on the **Data Properties** pane. The names of the arguments that appear in the outputs wrapper function are the same as the names found on the **Output ports** pane. The width of each array is the same as the output width specified for each output on the **Output ports** 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 the Simulink engine.
`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 4-33 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 4-31 for more information).
param0, p_width0, param1, p_width1, ... paramN, p_widthN	param0, param1, ...paramN are pointers to arrays containing the S-function parameters, where N is the number of parameters specified on the **Parameters** pane found on the **Data Properties** pane. p_width0, p_width1, ...p_widthN are the widths of the parameter arrays. If a parameter is a matrix, the width equals the product of the dimensions of the arrays. For example, the width of a 3-by-2 matrix parameter is 6. These arguments appear only if you specify parameters on the **Data Properties** pane.
y_width	Width of the array containing the S-function outputs. This argument appears in the generated code only if you specified -1 as the width of the S-function 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 inputs. This argument appears in the generated code only if you specified -1 as the width of the S-function input. If the input is a matrix, u_width is the product of the dimensions of the matrix.

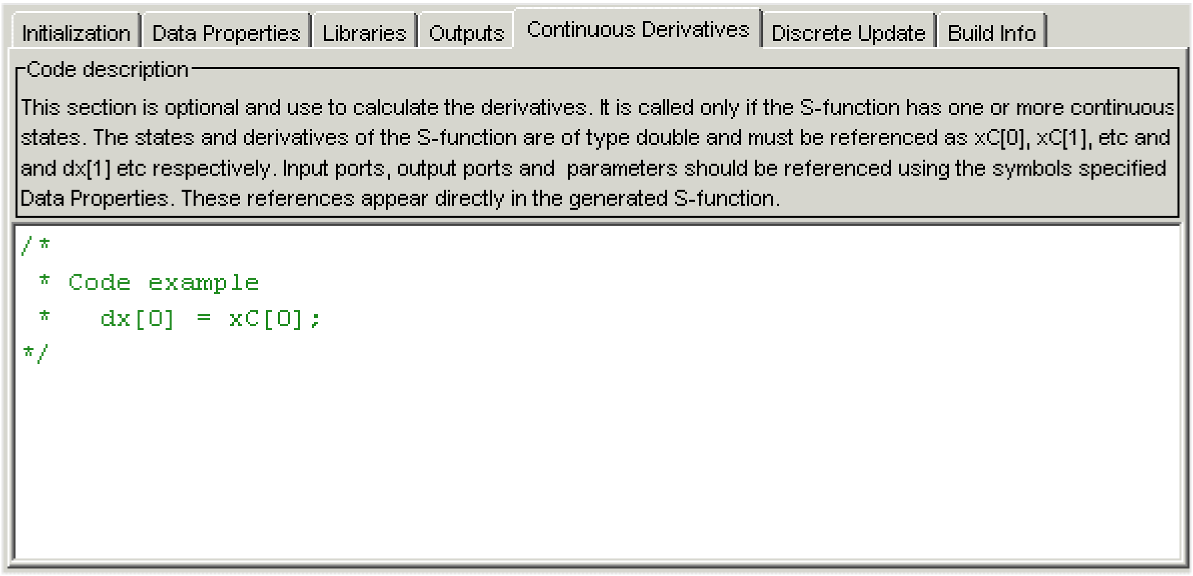
These arguments permit you to compute the output of the block as a function of its inputs and, optionally, its states and parameters. The code that you enter in this field can invoke external functions declared in the header files or external declarations on the **Libraries** pane. This allows you to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

Inputs are needed in the output function

Select this check box if the current values of the S-function inputs are used to compute its outputs. The Simulink engine uses this information to detect algebraic loops created by directly or indirectly connecting the S-function output to the S-function 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.



This section is optional and use to calculate the derivatives. It is called only if the S-function has one or more continuous states. The states and derivatives of the S-function are of type double and must be referenced as xC[0], xC[1], etc and and dx[1] etc respectively. Input ports, output ports and parameters should be referenced using the symbols specified Data Properties. These references appear directly in the generated S-function.

```
/*
 * Code example
 *   dx[0] = xC[0];
*/
```

Enter code to compute the derivatives of the 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. The Simulink engine calls the `mdlDerivatives` method at the end of each time step to obtain the derivatives of the continuous states (see "How the Simulink Engine Interacts with C S-Functions" on page 4-77). The Simulink solver numerically integrates the derivatives to determine the continuous states at the next time step. At the next time step, the engine passes the updated states back to the `mdlOutputs` method (see "Outputs Pane" on page 4-27).

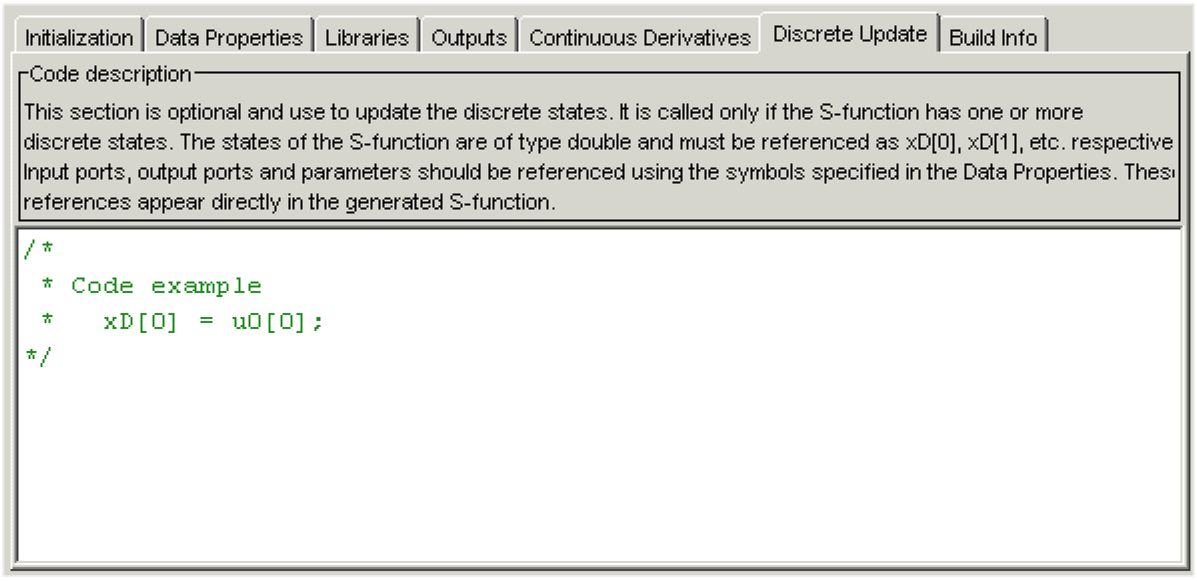
The `mdlDerivatives` callback method generated for the S-function passes the following arguments to the derivatives wrapper function:

- `u`
- `y`
- `dx`
- `xC`
- `param0`, `p_width0`, `param1`, `p_width1`, ... `paramN`, `p_widthN`
- `y_width`
- `u_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 "Outputs Pane" on page 4-27 for the meanings and usage of the other arguments. The arguments allow your code to compute derivatives as a function of the S-function 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.



This section is optional and use to update the discrete states. It is called only if the S-function has one or more discrete states. The states of the S-function are of type double and must be referenced as xD[0], xD[1], etc. respective Input ports, output ports and parameters should be referenced using the symbols specified in the Data Properties. Thes references appear directly in the generated S-function.

```
/*
 * Code example
 *   xD[0] = u0[0];
*/
```

Enter code to compute the values of the 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. The Simulink engine calls the `mdlUpdate` method at the end of each time step to obtain the values of the discrete states at the next time step (see "How the Simulink Engine Interacts with C S-Functions" on page 4-77). At the next time step, the engine passes the updated states back to the `mdlOutputs` method (see "Outputs Pane" on page 4-27).

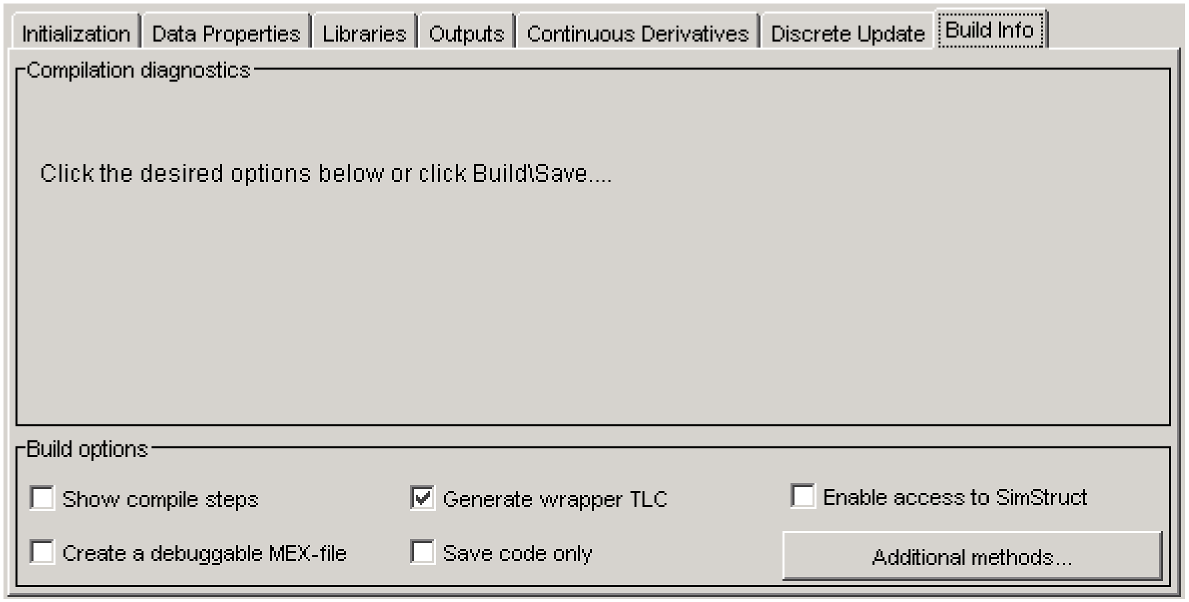
The `mdlUpdates` callback method generated for the S-function passes the following arguments to the updates wrapper function:

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

See "Outputs Pane" on page 4-27 for the meanings and usage of these arguments. Your code should use the `xD` (discrete states) variable to return the values of the discrete states that it computes. The arguments allow your code to compute the discrete states as functions of the S-function 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.



Compilation diagnostics

Click the desired options below or click Build\Save....

Build options

- [ ] Show compile steps
- [✓] Generate wrapper TLC
- [ ] Enable access to SimStruct
- [ ] Create a debuggable MEX-file
- [ ] Save code only

This pane contains the following fields.

Compilation diagnostics

Displays information as the S-Function Builder is generating the C source and executable files.

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

Selecting this option allows you to generate a TLC file. You need to generate a TLC file if you are running your model in Rapid Accelerator mode or generating Real-Time Workshop code from your model. Also, while it is not necessary for Accelerator mode simulations, the TLC file will generate code for the S-function and thus makes your model run faster in Accelerator mode.

Save code only

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

Enable access to SimStruct

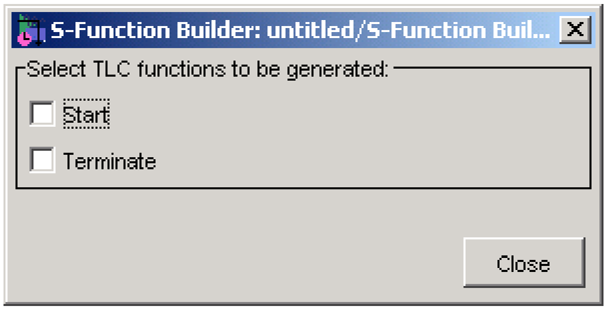
Makes the `SimStruct` (`S`) accessible to the wrapper functions that S-Function Builder generates. This enables you to use the `SimStruct` macros and functions with your code in the **Outputs**, **Continuous Derivatives**, and **Discrete Updates** panes. For example, with this option enabled, you can use macros such as `ssGetT` in code that computes the S-function outputs:

```
double t = ssGetT(S);
  if(t < 2 ) {
    y0[0] = u0[0];
  } else {
    y0[0]= 0.0;
  }
```

For a complete listing of `SimStruct` macros and functions, see Chapter 11, “SimStruct Functions — Alphabetical List” in the online documentation.

Additional methods

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


S-Function Builder: untitled/S-Function Buil...
Select TLC functions to be generated:
Start
Terminate
Close

Check the methods you want to add and click the **Close** button to include the methods in your TLC file. See "Block Target File Methods" in the Real-Time Workshop Target Language Compiler documentation for more information.

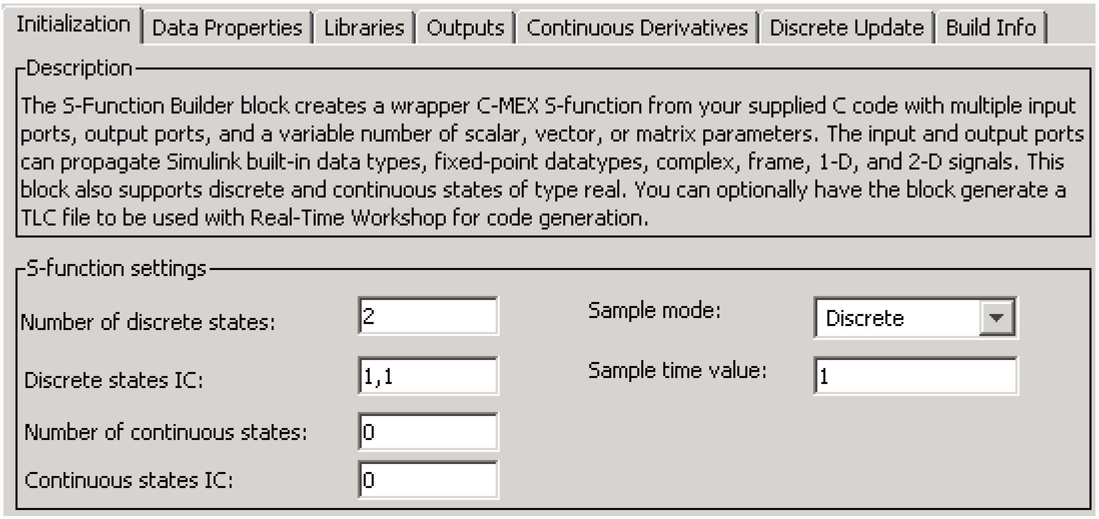
Example: Modeling a Two-Input/Two-Output System

The example `sfbuilder_example.mdl` shows how to use the S-Function Builder to model a two-input/two-output discrete state-space system with two states. In the example, the state-space matrices are parameters to the S-function and the S-function input and output are vectors. You can find a manually written version of the S-function in `dsfunc.c`.

Note You need to build the S-function before running the example model. To build the S-function, double-click on the S-Function Builder block in the model and click **Build** on the S-Function Builder dialog box that opens.

Initializing S-Function Settings

The **Initialization** pane specifies the number of discrete states and their initial conditions, as well as sets the sample time of the S-function. This example contains two discrete states, each initialized to 1, and a discrete sample mode with a sample time of 1.



Description

The S-Function Builder block creates a wrapper C-MEX S-function from your supplied C code with multiple input ports, output ports, and a variable number of scalar, vector, or matrix parameters. The input and output ports can propagate Simulink built-in data types, fixed-point datatypes, complex, frame, 1-D, and 2-D signals. This block also supports discrete and continuous states of type real. You can optionally have the block generate a TLC file to be used with Real-Time Workshop for code generation.

S-function settings

Number of discrete states: 2

Discrete states IC: 1,1

Number of continuous states: 0

Continuous states IC: 0

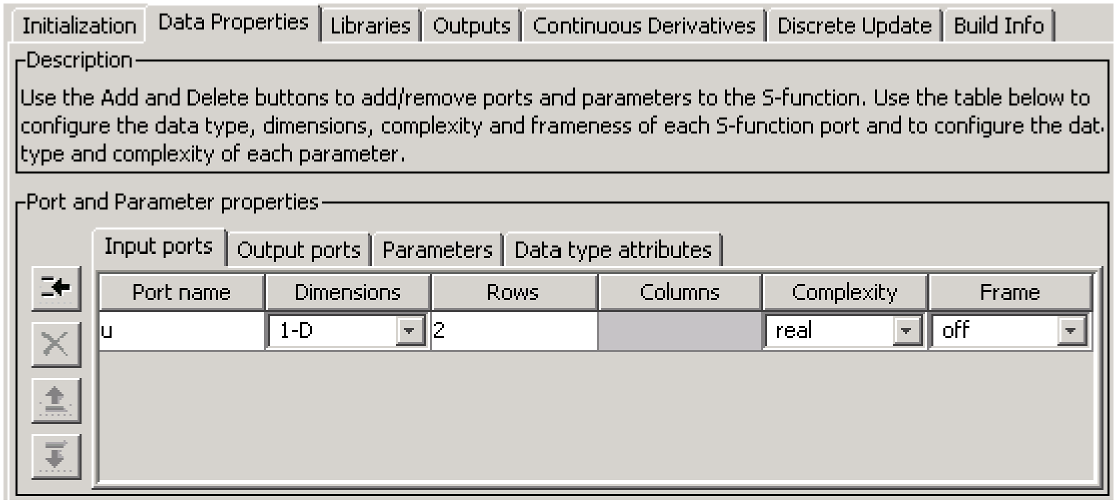
Sample mode: Discrete

Sample time value: 1

Initializing Inputs, Outputs, and Parameters

The **Data Properties** pane specifies the dimensions of the S-function input and output, as well as initializes the state-space matrices.

The **Input ports** pane defines the one S-function input port as a 1-D vector with two rows.


Initialization | Data Properties | Libraries | Outputs | Continuous Derivatives | Discrete Update | Build Info

Description

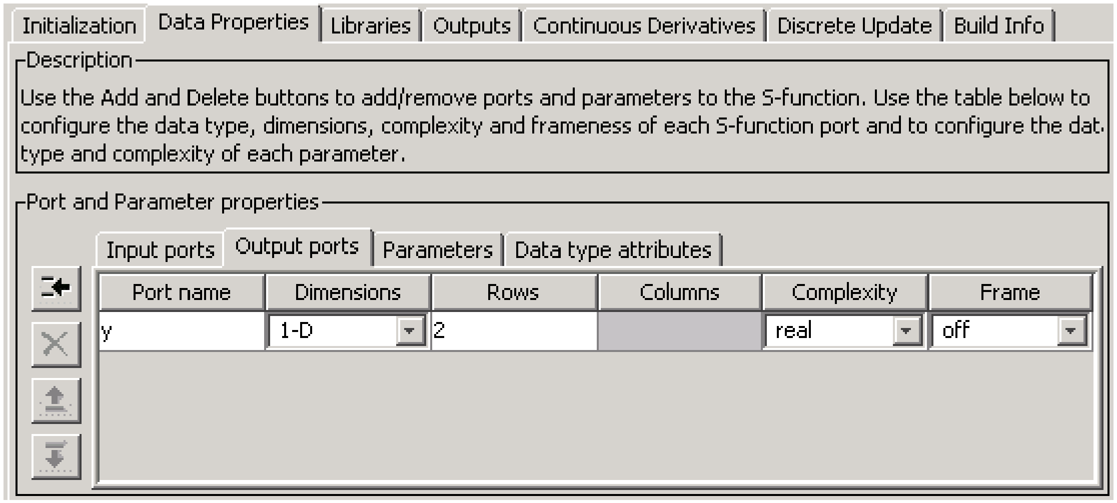
Use the Add and Delete buttons to add/remove ports and parameters to the S-function. Use the table below to configure the data type, dimensions, complexity and frameness of each S-function port and to configure the dat type and complexity of each parameter.

Port and Parameter properties

Input ports | Output ports | Parameters | Data type attributes

Port name	Dimensions	Rows	Columns	Complexity	Frame
u	1-D	2		real	off

The **Output ports** pane similarly defines the one S-function output port as a 1-D vector with two rows.


Initialization | Data Properties | Libraries | Outputs | Continuous Derivatives | Discrete Update | Build Info

Description

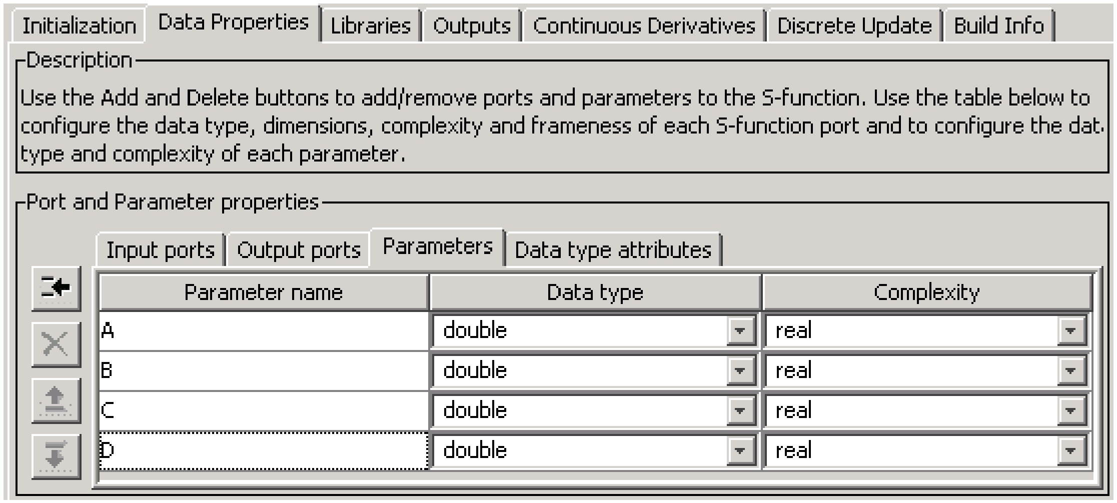
Use the Add and Delete buttons to add/remove ports and parameters to the S-function. Use the table below to configure the data type, dimensions, complexity and frameness of each S-function port and to configure the dat type and complexity of each parameter.

Port and Parameter properties

Input ports | Output ports | Parameters | Data type attributes

Port name	Dimensions	Rows	Columns	Complexity	Frame
y	1-D	2		real	off

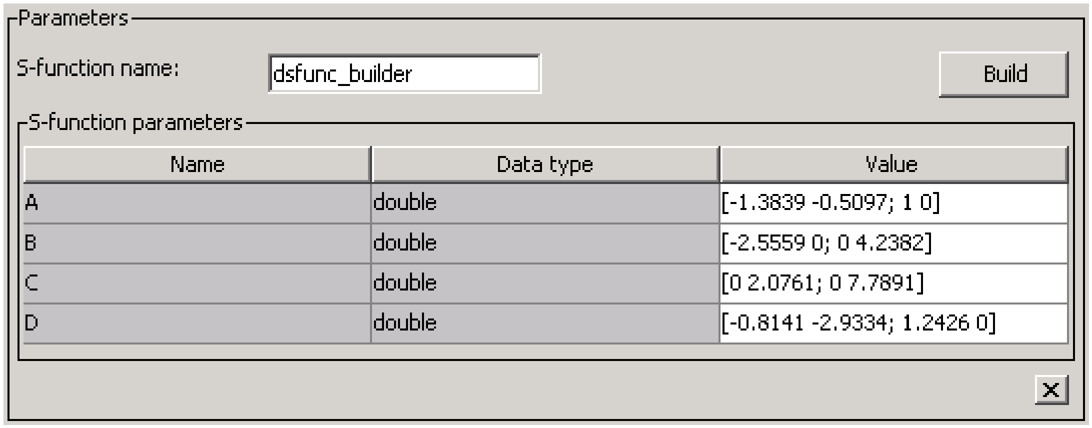
The **Parameters** pane defines four parameters, one for each of the four state-space matrices.



Use the Add and Delete buttons to add/remove ports and parameters to the S-function. Use the table below to configure the data type, dimensions, complexity and frameness of each S-function port and to configure the dat type and complexity of each parameter.

Parameter name	Data type	Complexity
A	double	real
B	double	real
C	double	real
D	double	real

The **S-function parameters** pane at the top of the S-Function Builder contains the actual values for the state-space matrices, entered as MATLAB expressions. In this example, each state-space parameter is a two-by-two matrix. Alternatively, you could store the state-space matrices in variables in the MATLAB workspace and enter the variable names into the **Value** field for each parameter.

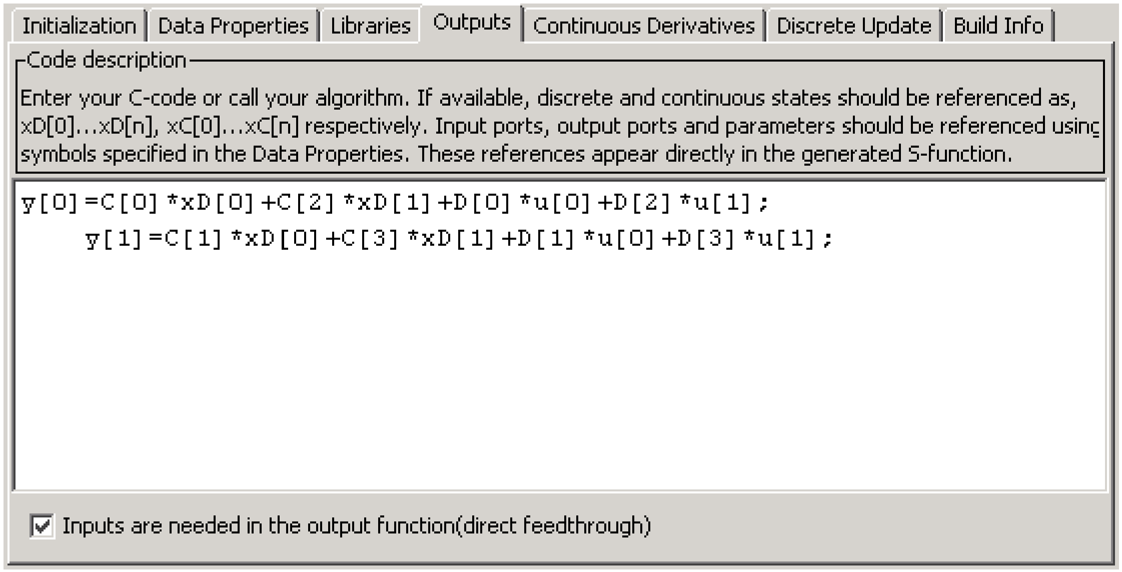


S-function name: dsfunc_builder

Name	Data type	Value
A	double	[-1.3839 -0.5097; 1 0]
B	double	[-2.5559 0; 0 4.2382]
C	double	[0 2.0761; 0 7.7891]
D	double	[-0.8141 -2.9334; 1.2426 0]

Defining the Output Method

The **Outputs** pane calculates the S-function output as a function of the input and state vectors and the state-space matrices. In the outputs code, reference S-function parameters using the parameter names defined on the **Data Properties — Parameters** pane. Index into 2-D matrices using a scalar index, keeping in mind that S-functions use zero-based indexing. For example, to access the element `C(2,1)` in the S-function parameter `C`, use `C[1]`in the S-function code.



Enter your C-code or call your algorithm. If available, discrete and continuous states should be referenced as, xD[0]...xD[n], xC[0]...xC[n] respectively. Input ports, output ports and parameters should be referenced using symbols specified in the Data Properties. These references appear directly in the generated S-function.

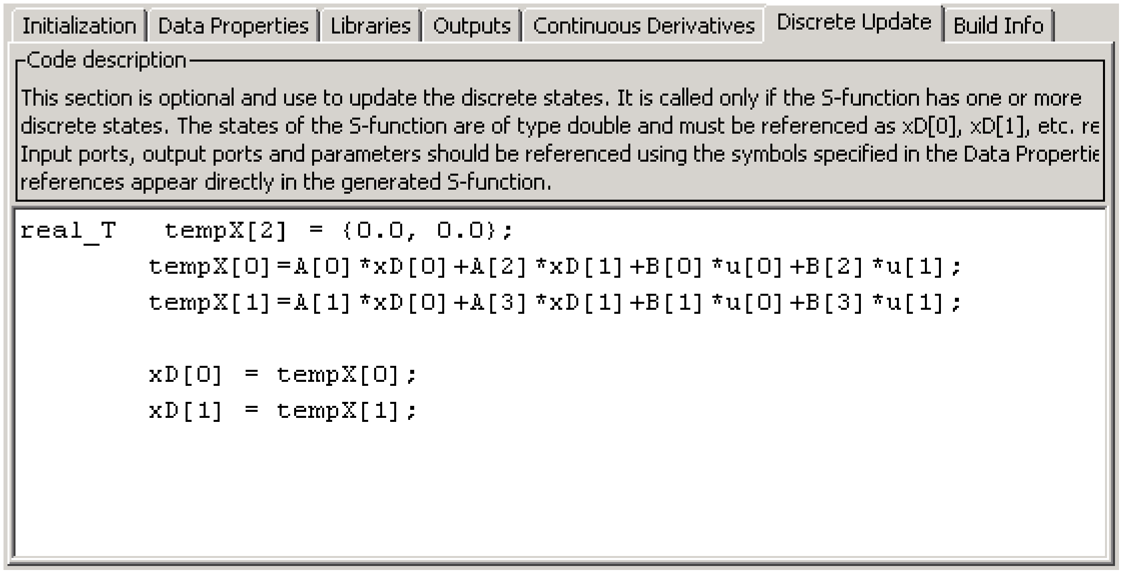
```
y[0]=C[0]*xD[0]+C[2]*xD[1]+D[0]*u[0]+D[2]*u[1];
    y[1]=C[1]*xD[0]+C[3]*xD[1]+D[1]*u[0]+D[3]*u[1];
```

[✓] Inputs are needed in the output function(direct feedthrough)

The **Outputs** pane also selects the **Inputs are needed in the output function (direct feedthrough)** option since this state-space model has a nonzero `D` matrix.

Defining the Discrete Update Method

The **Discrete Update** pane updates the discrete states. As with the outputs code, use the S-function parameter names and index into 2-D matrices using a scalar index, keeping in mind that S-functions use zero-based indexing. For example, to access the element `A(2,1)` in the S-function parameter `A`, use `A[1]`in the S-function code. The variable `xD` stores the final values of the discrete states.



Code description

This section is optional and use to update the discrete states. It is called only if the S-function has one or more discrete states. The states of the S-function are of type double and must be referenced as xD[0], xD[1], etc. re Input ports, output ports and parameters should be referenced using the symbols specified in the Data Propertie references appear directly in the generated S-function.

```
real_T   tempX[2] = {0.0, 0.0};
        tempX[0]=A[0]*xD[0]+A[2]*xD[1]+B[0]*u[0]+B[2]*u[1];
        tempX[1]=A[1]*xD[0]+A[3]*xD[1]+B[1]*u[0]+B[3]*u[1];

        xD[0] = tempX[0];
        xD[1] = tempX[1];
```

Building the State-Space Example

Click the **Build** button on the S-Function Builder to create an executable for this S-function. You can now run the model and compare the output to the original discrete state-space S-function contained in `sfcndemo_dsfunc.mdl`.

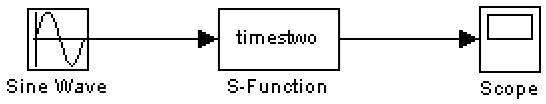
Example of a Basic C MEX S-Function

In this section...

Introducing an Example of a Basic C MEX S-Function

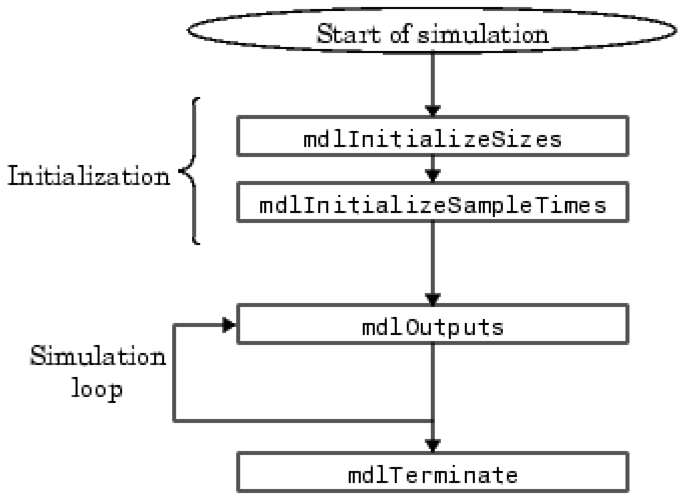
This section presents an example of a C MEX S-function that you can use as a model for creating simple C S-functions. The example S-function `timestwo.c` outputs twice its input.

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. A description of the code is provided after the example.

```
#define S_FUNCTION_NAME timestwo /* Defines and Includes */
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

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

    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){}


/* Simulink/Real-Time Workshop Interface */

#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 method implementations
- Simulink (or Real-Time Workshop) product interfaces

Defines and Includes

The example starts with the following `define` statements.

```
#define S_FUNCTION_NAME  timestwo
#define S_FUNCTION_LEVEL 2
```

The first `define` statement specifies the name of the S-function (`timestwo`). The second `define` statement 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 4-101).

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. (Go to code in `timestwo.c` example.)

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

The `simstruc.h` file defines a data structure, called the `SimStruct`, that the Simulink engine 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 Reference").

Callback Method Implementations

The next part of the `timestwo` S-function contains implementations of required callback methods.

mdlInitializeSizes

The Simulink engine calls `mdlInitializeSizes` to inquire about the number of input and output ports, sizes of the ports, and any other information (such as the number of states) needed by the S-function. (Go to code in `timestwo.c` example.)

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

- Zero parameters

  Therefore, the **S-function parameters** field of the S-Function Block Parameters dialog box must be empty. If it contains any parameters, the engine reports a parameter mismatch.

- One input port and one output port

  The widths of the input and output ports are dynamically sized. This tells the engine that the S-function can accept an input signal of any width. By default, the widths of dynamically sized input and output port are equal when the S-function has only one input and output port.

- One sample time

  The `mdlInitializeSampleTimes` callback method specifies the actual value of the sample time.

- Exception free code

  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 is not interacting with the MATLAB environment, you can safely specify this option. For more details, see "How the Simulink Engine Interacts with C S-Functions" on page 4-77.

mdlInitializeSampleTimes

The Simulink engine 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, `INHERITED_SAMPLE_TIME`. (Go to code in `timestwo.c` example.)

mdlOutputs

The engine calls `mdlOutputs` at each time step to calculate the block outputs. The `timestwo` implementation of `mdlOutputs` multiplies the input signal by 2 and writes the answer to the output.

The line:

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

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

```
*uPtrs[i]
```

For more details on accessing input signals, see “Accessing Signals Using Pointers” on page 4-86.

The line:

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

accesses the output signal. The `ssGetOutputPortRealSignal` macro returns a pointer to an array containing the block outputs.

The line:

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

obtains the width of the signal passing through the block. The S-function loops over the inputs to compute the outputs. (Go to code in `timestwo.c` example.)

mdlTerminate

The engine calls `mdlTerminate` to provide the S-function with an opportunity to perform tasks at the end of the simulation. This is a mandatory S-function routine. The `timestwo` S-function does not perform any termination actions, and this routine is empty. (Go to code in `timestwo.c` example.)

Simulink/Real-Time Workshop Interfaces

At the end of the S-function, include the following code to attach your S-function to either the Simulink or Real-Time Workshop products.

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

This trailer is required at the end of every S-function. If it is omitted, any attempt to compile your S-function will abort with a `failure during build of exports file` error message. (Go to code in `timestwo.c` example.)

Building the Timestwo Example

To compile this S-function, 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 the Simulink software to use.

The resulting executable is referred to as a MEX S-function, where MEX stands for "MATLAB Executable." The MEX-file extension varies from platform to platform. For example, on a 32–bit Microsoft® Windows® system, the MEX-file extension is `.mexw32`.

Templates for C S-Functions

In this section...

About the Templates for C S-Functions

Use one of the provided C MEX S-function templates as a starting point for creating your own S-function. 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 *`matlabroot`*`/simulink/src`, 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 S-Function Block Parameters dialog box). 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 4-101. In addition, the code specifies that you are using the Level-2 S-function format.

Note All S-functions from Simulink version 1.3 through version 2.1 are considered to be Level-1 S-functions. They are compatible with newer versions of the software, but we recommend that you write new S-functions in the Level-2 format.

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

Header Files Included by simstruc.h When Compiling as a MEX-File

Header File	Description
`matlabroot/extern/include/tmwtypes.h`	General data types, e.g., `real_T`
`matlabroot/simulink/include/simstruc_types.h`	`SimStruct` data types, e.g., `BuiltInDTypeId`
`matlabroot/extern/include/mex.h`	MATLAB MEX-file API routines to interface MEX-files with the MATLAB environment
`matlabroot/extern/include/matrix.h`	MATLAB External Interface API routines to query and manipulate MATLAB matrices

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

Header Files Included by simstruc.h When Used by the Real-Time Workshop Product

Header File	Description
matlabroot/extern/include/tmwtypes.h	General types, e.g., `real_T`
matlabroot/simulink/include/simstruc_types.h	`SimStruct` data types, e.g., `BuiltInDTypeId`
matlabroot/rtw/c/src/rt_matrx.h	Macros for MATLAB API routines

Callback Methods That an S-Function Must Implement

Your S-function must implement the following functions (see “Writing Callback Methods” on page 4-90):

- `mdlInitializeSizes` specifies the sizes of various parameters in the `SimStruct`, such as the number of output ports for the block.
- `mdlInitializeSampleTimes` specifies the sample time(s) of the block.
- `mdlOutputs` calculates the output of the block.
- `mdlTerminate` performs any actions required at the termination of the simulation. If no actions are required, this function can be implemented as a stub.

Statements Required at the Bottom of S-Functions

Your S-function must include the following trailer code at the end of the 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 with the Real-Time Workshop product to produce a standalone 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 SimStruct data structure and its 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. Any SimStruct associated with an S-function is a *child* SimStruct.

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

Data Types in S-Functions

The file *matlabroot*/extern/include/tmwtypes.h is a C language header file that defines a set of data types used in the S-function template and in the SimStruct. These data types, such as real_T, uint32_T, etc., provide a way to switch between different data types for 16, 32, and 64 bit systems, allowing greater platform independence and flexibility.

S-functions are not required to use these data types. For example, you can edit the example *matlabroot*/simulink/src/csfunc.c and change real_T to double and int_T to int. If you compile and simulate the S-function, the results will be identical to the results using the previous data types.

Compiling C S-Functions

Your S-function can be compiled in one of three modes, defined either by the mex command or by the Real-Time Workshop product when the S-function is built:

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

The build process you use automatically defines the mode for your S-function.

Integrating Existing C Functions into Simulink Models with the Legacy Code Tool

In this section...

Overview

You can integrate existing C (or C++) functions—for example, device drivers, lookup tables, and general functions and interfaces—into Simulink models by using the Legacy Code Tool. Using specifications that you supply as M-code, the tool transforms existing functions into C MEX S-functions that you can include in Simulink models. If you use the Real-Time Workshop product to generate code from a model, the Legacy Code Tool can insert an appropriate call to your C function into the generated code (for details, see "Automating the Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool" in the Real-Time Workshop documentation).

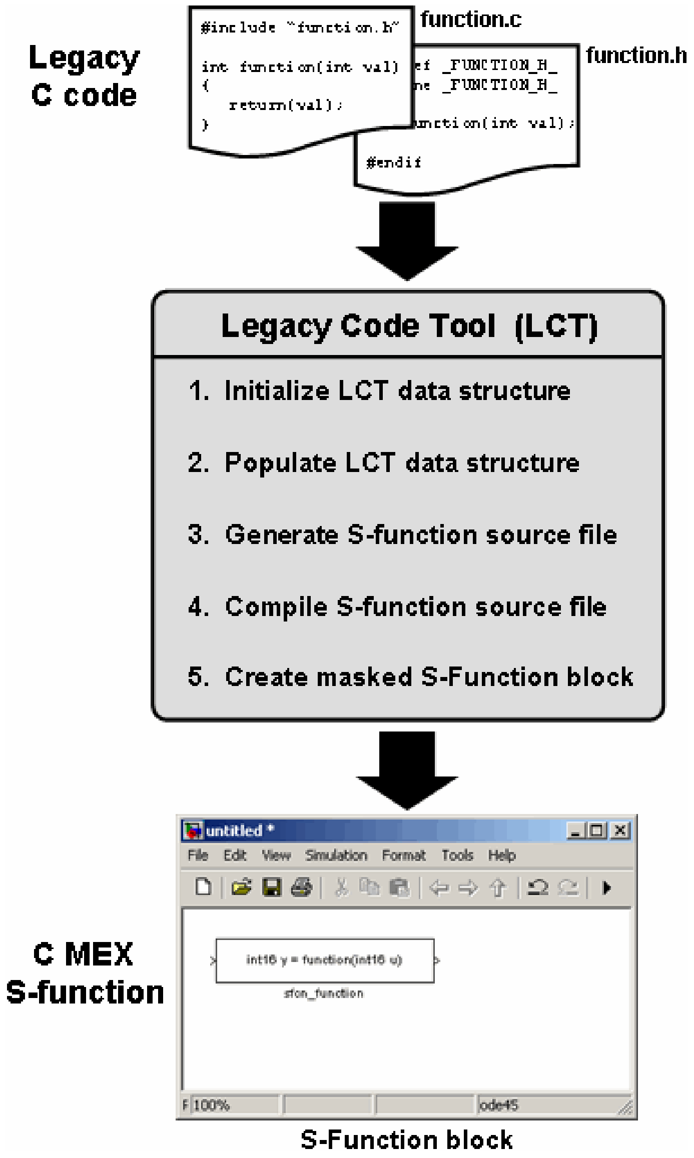
In comparison to using the S-Function Builder or writing an S-function, the Legacy Code Tool can be easier to use and it generates optimized code (does not generate wrapper code) often required by embedded systems. However, you should consider one of the alternate approaches for a hybrid system, such as a system that includes a plant and controller, or a system component written in a language other than C or C++. Alternative approaches are more flexible in that they support more features and programming languages.

To interact with the Legacy Code Tool, you

- Use a Legacy Code Tool data structure to specify
  - A name for the S-function
  - Specifications for the existing C functions
  - Files and paths required for compilation
  - Options for the generated S-function
- Use the `legacy_code` function to
  - Initialize the Legacy Code Tool data structure for a given C function
  - Generate an S-function for use during simulation
  - Compile and link the generated S-function into a dynamically loadable executable
  - Generate a masked S-function block for calling the generated S-function
  - Generate a TLC block file and, if necessary, an `rtwmakecfg.m` file for code generation (Real-Time Workshop product license required)

Note Before you can use `legacy_code`, you must ensure that a C compiler is set up for your MATLAB installation. If you need to set up a compiler, enter the command `mex -setup` in the MATLAB Command Window.

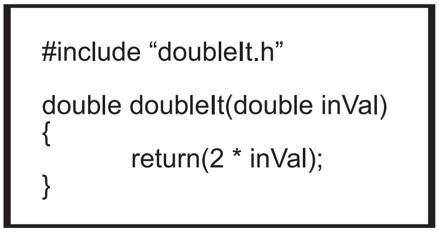
The following diagram illustrates a general procedure for using the Legacy Code Tool. "Example of Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" on page 4-58 provides an example that uses the Legacy Code Tool to transform an existing C function into a C MEX S-function.


Legacy C code
function.c
function.h
#include "function.h"
int function(int val)
{
return(val);
}
_FUNCTION_H_
_FUNCTION_H_
function(int val);
#endif
Legacy Code Tool (LCT)
1. Initialize LCT data structure
2. Populate LCT data structure
3. Generate S-function source file
4. Compile S-function source file
5. Create masked S-Function block
C MEX S-function
untitled *
File Edit View Simulation Format Tools Help
int16 y = function(int16 u)
sfcn_function
100%
ode45
S-Function block

If you have a Real-Time Workshop product license, see "Automating the Generation of Files for Fully Inlined S-Functions Using Legacy Code Tool" in the Real-Time Workshop documentation for information on using the Legacy Code Tool for code generation.

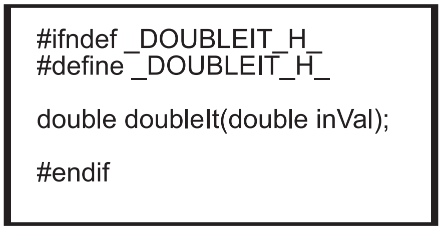
Example of Integrating Existing C Functions into Simulink Models with the Legacy Code Tool

Suppose you have an existing C function that outputs the value of its floating-point input multiplied by two. The function is defined in a source file named `doubleIt.c`, and its declaration exists in a header file named `doubleIt.h` as shown here.



```
#include “doubleIt.h”

double doubleIt(double inVal)
{
        return(2 * inVal);
}
```

doubleIt.c



```
#ifndef _DOUBLEIT_H_
#define _DOUBLEIT_H_

double doubleIt(double inVal);

#endif
```

doubleIt.h

To use the Legacy Code Tool to incorporate this C function into a Simulink model as a C MEX S-function:

1 Use the `legacy_code` function to initialize a MATLAB structure with fields that represent Legacy Code Tool properties. For example, create a Legacy Code Tool data structure named `def` by entering the following command at the MATLAB command prompt:

```
def = legacy_code('initialize')
```

The Legacy Code Tool data structure named `def` displays its fields in the MATLAB Command Window as shown here:

```
def =

                 SFunctionName: ''
   InitializeConditionsFcnSpec: ''
                 OutputFcnSpec: ''
                  StartFcnSpec: ''
              TerminateFcnSpec: ''
```

```
      HeaderFiles: {}
      SourceFiles: {}
     HostLibFiles: {}
   TargetLibFiles: {}
         IncPaths: {}
         SrcPaths: {}
         LibPaths: {}
       SampleTime: 'inherited'
          Options: [1x1 struct]
```

2 Specify appropriate values for fields in the Legacy Code Tool data structure to identify properties of the existing C function. For example, specify the C function source and header filenames by entering the following commands at the MATLAB command prompt:

```
def.SourceFiles = {'doubleIt.c'};
def.HeaderFiles = {'doubleIt.h'};
```

You must also specify information about the S-function that the Legacy Code Tool produces from the C code. For example, specify a name for the S-function and its output function declaration by entering:

```
def.SFunctionName = 'ex_sfun_doubleit';
def.OutputFcnSpec = 'double y1 = doubleIt(double u1)';
```

See the `legacy_code` reference page for information about the various data structure fields. For more information about assigning values to fields in a structure, see “Structures” in the MATLAB documentation.

3 Use the `legacy_code` function to generate an S-function source file from the existing C function. At the MATLAB command prompt, type:

```
legacy_code('sfcn_cmex_generate', def);
```

The Legacy Code Tool uses the information specified in `def` to create the S-function source file named `ex_sfun_doubleit.c` in the current MATLAB directory.

4 Make sure a C compiler is set up for your MATLAB installation. If you need to set up a compiler, enter the command `mex -setup` in the MATLAB Command Window.

5 Use the `legacy_code` function to compile and link the S-function source file into a dynamically loadable executable that the Simulink software can use. At the MATLAB command prompt, type:

```
legacy_code('compile', def);
```

The following messages appear in the MATLAB Command Window:

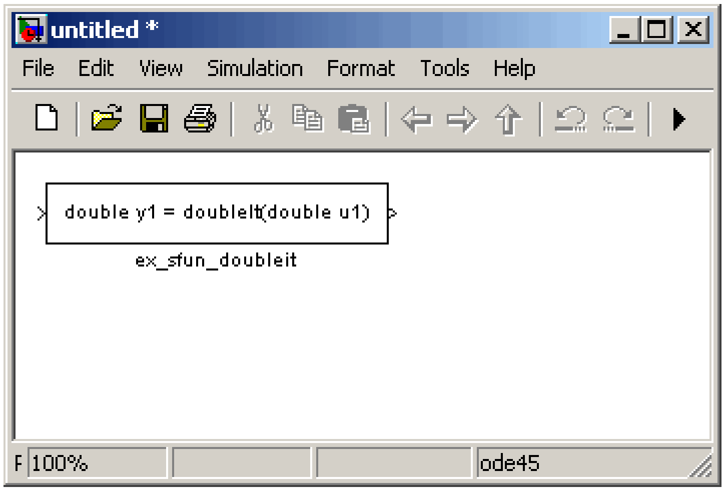
```
### Start Compiling ex_sfun_doubleit
    mex('ex_sfun_doubleit.c', 'd:\work\lct_demos\doubleIt.c',
       '-Id:\work\lct\lct_demos')
### Finish Compiling ex_sfun_doubleit
### Exit
```

On a 32-bit Microsoft Windows system, the resulting S-function executable is named `ex_sfun_doubleit.mexw32`.

6 Use the `legacy_code` function to insert a masked S-Function block into a Simulink model. The Legacy Code Tool configures the block to use the C MEX S-function created in the previous step. Also, the tool masks the block such that it displays the value of its `OutputFcnSpec` property (see the description of the `legacy_code` function). For example, create a new model containing a masked S-Function block by issuing the following command at the MATLAB command prompt:

```
legacy_code('slblock_generate', def);
```

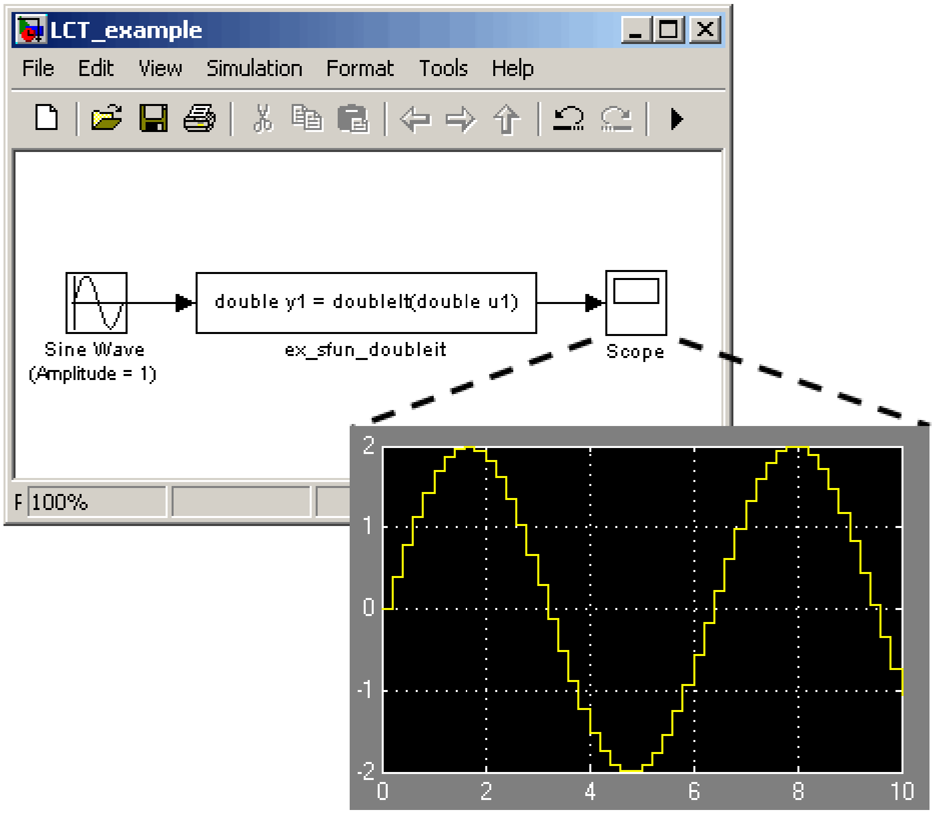
The block appears in an empty model editor window as shown here:



double y1 = doubleIt(double u1)

ex_sfun_doubleit

The following Simulink model demonstrates that the C MEX S-function produced by the Legacy Code Tool behaves like the C function `doubleIt`. In particular, the S-Function block named `ex_sfun_doubleit` returns the value of its floating-point input multiplied by two.


LCT_example
File Edit View Simulation Format Tools Help
double y1 = doubleIt(double u1)
Sine Wave
(Amplitude = 1)
ex_sfun_doubleit
Scope
100%
2
1
0
-1
-2
0
2
4
6
8
10

Registering Legacy Code Tool Data Structures

The first step to using the Legacy Code Tool is to register one or more MATLAB structures with fields that represent properties of the existing C code and the S-function being generated. The registration process is flexible. You can choose to set up resources and initiate registration in a variety of ways, including

- Placing all required header and source files in the current working directory or in a hierarchical directory structure
- Generating and placing one or more S-functions in the current working directory

- Having one or more registration files in the same directory

To register a Legacy Code Tool data structure:

1 Use the `legacy_code` function, specifying `'initialize'` as the first argument.

```
lct_spec = legacy_code('initialize')
```

The Legacy Code Tool data structure named `lct_spec` displays its fields in the MATLAB Command Window as shown below:

```
lct_spec =

              SFunctionName: ''
InitializeConditionsFcnSpec: ''
              OutputFcnSpec: ''
               StartFcnSpec: ''
           TerminateFcnSpec: ''
                HeaderFiles: {}
                SourceFiles: {}
               HostLibFiles: {}
             TargetLibFiles: {}
                   IncPaths: {}
                   SrcPaths: {}
                   LibPaths: {}
                 SampleTime: 'inherited'
                    Options: [1x1 struct]
```

2 Define values for the data structure fields (properties) that apply to your existing C function and the S-function you intend to generate. Minimally, you must specify

- Source and header files for the existing C function (`SourceFiles` and `HeaderFiles`)
- A name for the S-function (`SFunctionName`)
- At least one function specification for the S-function (`InitializeConditionsFcnSpec`, `OutputFcnSpec`, `StartFcnSpec`, `TerminateFcnSpec`)

For a complete list and descriptions of the fields in the structure, see the `legacy_code` function reference page.

If you define fields that specify compilation resources and you specify relative paths, the Legacy Code Tool searches for the resources relative to the following directories, in the following order:

1. Current working directory
2. C-MEX S-function directory, if different than the current working directory
3. Directories you specify
    - `IncPaths` for header files
    - `SrcPaths` for source files
    - `LibPaths` for target and host libraries
4. Directories on the MATLAB search path, excluding toolbox directories

Declaring Legacy Code Tool Function Specifications

The `InitializeConditionsFcnSpec`, `OutputFcnSpec`, `StartFcnSpec`, and `TerminateFcnSpec` fields defined in the Legacy Code Tool data structure (see the description of the `legacy_code` function) require string values that adhere to a specific syntax format. The required syntax format enables the Legacy Code Tool to map the return value and arguments of an existing C function to the return value, inputs, outputs, parameters, and work vectors of the S-function that the tool generates.

General syntax

```
return-spec = function-name(argument-spec)
```

For example, the following string specifies a function named `doubleIt` with return specification `double y1` and input argument specification `double u1`.

```
def.OutputFcnSpec = 'double y1 = doubleIt(double u1)';
```

For more detail on declaring function specifications, see

- "Return Specification" on page 4-65

- “Function Name” on page 4-65
- “Argument Specification” on page 4-66
- “Supported Data Types” on page 4-69
- “Function Specification Rules” on page 4-70

Return Specification

The return specification defines the data type and variable name for the return value of the existing C function.

```
return-type return-variable
```

return-type	A data type listed in “Supported Data Types” on page 4-69.
return-variable	Token of the form `y1`, `y2`, `...`, `y`*n*, where *n* is the total number of output arguments.

If the function does not return a value, you can omit the return specification or specify it as `void`.

The following table shows valid function specification syntax for an integer return value. Use the table to identify the syntax you should use for your C function prototype.

Return Type	C Function Prototype	Legacy Code Tool Function Specification
No return value	`void myfunction(...)`	`void myfunction(...)`
Scalar value	`int = myfunction(...)`	`int16 y1 = myfunction(...)`

Function Name

The function name that you specify must be the same as your existing C function name.

For example, consider the following C function prototype:

```
float doubleIt(float inVal);
```

In this case, the function name in the Legacy Code Tool function specification must be `doubleIt`.

You should not specify the name of a C macro. If you must, set the field `Options.isMacro` to 1 to ensure that the generated code remains safe in the event that expression folding is enabled.

Argument Specification

The argument specification defines one or more data type and token pairs that represent the input, output, parameter, and work vector arguments of the existing C function. The function input and output arguments map to block input and output ports and parameters map to workspace parameters.

```
argument-type argument-token
```

`argument-type`	A data type listed in "Supported Data Types" on page 4-69.
`argument-token`	Token of one of the following forms: • Input — `u1`, `u2`, `...`, `u`*n*, where *n* is the total number of input arguments • Output — `y1`, `y2`, `...`, `y`*n*, where *n* is the total number of output arguments • Parameter — `p1`, `p2`, `...`, `p`*n*, where *n* is the total number of parameter arguments • Work vectors (persistent memory) — `work1`, `work2`, `...`, `work`*n*, where *n* is the total number of work vector arguments

If the function has no arguments, you can omit the argument specification or specify it as `void`.

Consider the following C function prototype:

```
float powerIt(float inVal, int exponent);
```

To generate an S-function that calls the preceding function at each time step, you would set the Legacy Code Tool data structure field `OutputFcnSpec` to the following string:

```
'double y1 = powerIt(double u1, int16 p1)'
```

Using this function specification, the Legacy Code Tool maps the following:

Return Value or Argument...	of C Type...	To Token...	of Data Type...
Return value	`float`	`y1`	`double`
`inVal`	`float`	`u1`	`double`
`exponent`	`int`	`p1`	`int16`

The following table shows valid function specification syntax for arguments of type integer. Use the table to identify and then adapt the syntax you should use for your C function prototype.

Argument Type	C Function Prototype	Legacy Code Tool Function Specification
Input Arguments		
No arguments	`function(void)`	`function(void)`
Scalar pass by value	`function(int in1)`	`function(int16 u1)`
Scalar pass by pointer	`function(int *in1)`	`function(int16 u1[1])`
Fixed vector	`function(int in1[10])` or `function(int *in1)`	`function(int16 u1[10])`
Variable vector	`function(int in1[])` or `function(int *in1)`	`function(int16 u1[])`
Fixed matrix	`function(int in1[15])` or `function(int in1[])` or `function(int *in1)`	`function(int16 u1[3][5])`
Variable matrix	`function(int in1[])` or `function(int *in1)`	`function(int16 u1[][])`
Output Arguments		

Argument Type	C Function Prototype	Legacy Code Tool Function Specification
Scalar pointer	`function(int *y1)`	`function(int16 y1[1])`
Fixed vector	`function(int y1[10])` or `function(int *y1)`	`function(int16 y1[10])`
Fixed matrix	`function(int y1[15])` or `function(int y1[])` or `function(int *y1)`	`function(int16 y1[3][5])`
Parameter Arguments		
Scalar pass by value	`function(int p1)`	`function(int16 p1)`
Scalar pass by pointer	`function(int *p1)`	`function(int16 p1[1])`
Fixed vector	`function(int p1[10])` or `function(int *p1)`	`function(int16 p1[10])`
Variable vector	`function(int p1[])` or `function(int *p1)`	`function(int16 p1[])`
Fixed matrix	`function(int p1[15])` or `function(int p1[])` or `function(int *p1)`	`function(int16 p1[3][5])`
Variable matrix	`function(int p1[])` or `function(int *p1)`	`function(int16 p1[][])`
Work Vector Arguments		
Scalar pointer	`function(int *work1)` `function(void *work1)` `function(void **work1)`	`function(int16 work1[1])` `void function(void *work1)` `void function(void **work1)`
Fixed vector	`function(int work1[10])` or `function(int *work1)`	`function(int16 work1[10])`
Fixed matrix	`function(int work1[15])` or `function(int work1[])` or `function(int *work1)`	`function(int16 work1[3][5])`

Supported Data Types

Data Type	Supported for Input and Output?	Supported for Parameters?	Supported for Work Vectors?
"Data Types Supported by Simulink"	Yes	Yes	Yes
`Simulink.Bus`[1] (scalar only)	Yes	N/A	Yes
`Simulink.NumericType`[2]	Yes	Yes	Yes
`Simulink.AliasType`[1]	Yes	Yes	Yes
Fixed-point[3]	Yes	Yes	Yes
Fi objects	N/A	Yes	N/A
Complex numbers[4]	Yes	Yes	Yes
1-D array	Yes	Yes	Yes
2-D array[5]	Yes	Yes	Yes
n-D array[6]	Yes	Yes	Yes
void *	No	No	Yes
void **	No	No	Yes

1 You must supply the header file that declares the structure of the bus or the header file that defines the data type with the same name as an alias. The structure of the bus declared in the header file must match the structure of the bus object (for example, the number and order of elements, data types and widths of elements, and so on). For an example, see `sldemo_lct_bus`.

2 You must supply the header file that defines the data type only if the numeric data type is also an alias.

3 You must declare the data as a `Simulink.NumericType` object (unspecified scaling is not supported). For examples, see `sldemo_lct_fixpt_signals` and `sldemo_lct_fixpt_params`.

4 Limited to use with Simulink built-in data types. To specify a complex data type, enclose the built-in data type within angle brackets (<>) and prepend the word `complex` (for example, `complex<double>`). For an example, see `sldemo_lct_cplxgain`.

5 The MATLAB, Simulink, and Real-Time Workshop products store two-dimensional matrix data in column-major format as a vector. If your external function code is written for row-major data, transpose the matrix data in the MATLAB environment.

6 For a multidimensional signal, you can use the `size` function to determine the number of elements in the signal. For examples, see `sldemo_lct_lut` and `sldemo_lct_ndarray`.

For more information, see “Data Types Supported by Simulink”in *Using Simulink*.

Function Specification Rules

Legacy Code Tool function specifications must adhere to the following rules:

- If an argument is not scalar, you must pass the argument by reference.
- The function must not change the value of input arguments.
- The function’s return value cannot be a pointer.
- Function specifications you define for the `StartFcnSpec`, `InitializeConditionsFcnSpec`, or `TerminateFcnSpec` cannot access input or output arguments.
- The numbering of input, output, parameter, and work vector argument tokens must start at 1 and increase monotonically.
- For a given Legacy Code Tool data structure, the data type and size of input, output, parameter, and work vector arguments must be the same across function specifications for `StartFcnSpec`, `InitializeConditionsFcnSpec`, `OutputFcnSpec`, and `TerminateFcnSpec`.
- You can use the `size` function to
  - Get the size of any input, output, parameter, or work vector argument and pass the size as input to the legacy function
  - Specify the input argument dimensions as a function of other parameter argument dimensions

- Specify the output or work vector argument dimensions as a function of other input or parameter argument dimensions

Consider the following example, which demonstrates both uses of the function:

```
def.OutputFcnSpec=
'void foo(double p1[][], double u1[size(p1,2)], double y1[size(u1,1)], ...
double work1[size(u1,1)], int32 size(u1,1))'
```

- `p1` is a two-dimensional parameter that is sized dynamically
- `u1` is a one-dimensional vector with the same number of elements as the second dimension of `p1`
- `y1` is a one-dimensional vector with the same number of element as `u1`
- `work1` is a one-dimensional vector with the same number of element as `u1`
- `int32 size(u1,1)` returns the number of elements in the vector `u1` as the fifth input argument

Generating and Compiling the S-Functions

After you register a Legacy Code Tool data structure for an existing C function, use the `legacy_code` function as explained below to generate, compile, and link the S-function.

1. Generate a C MEX S-function based on the information defined in the structure. Call `legacy_code` with `'sfcn_cmex_generate'` as the first argument and the name of the data structure as the second argument.

   ```
   legacy_code('sfcn_cmex_generate', lct_spec);
   ```

2. Make sure a C compiler is set up for your MATLAB installation. If you need to set up a compiler, enter the command `mex -setup` in the MATLAB Command Window.

3. Compile and link the S-function. This step assumes that a C compiler is set up for your MATLAB installation. Call `legacy_code` with `'compile'` as the first argument and the name of the data structure as the second argument.

   ```
   legacy_code('compile', lct_spec);
   ```

Informational messages similar to the following appear in the MATLAB Command Window and a dynamically loadable executable results. On a 32-bit Windows system, the Simulink software names the file `ex_sfun_doubleit.mexw32`.

```
### Start Compiling ex_sfun_doubleit
mex  ex_sfun_doubleit.c -Id:\work\lct\lct_demos
### Finish Compiling ex_sfun_doubleit
### Exit
```

As a convenience, you can generate, compile, and link the S-function in a single step by calling `legacy_code` with the string `'generate_for_sim'`. The function also generates a TLC file for accelerated simulations, if the `Options.useTlcWithAccel` field of the Legacy Code Tool data structure is set to 1.

Once you have generated a dynamically loadable executable, you or others can use it in a model by adding an S-Function block that specifies the compiled S-function.

Generating a Masked S-Function Block for Calling a Generated S-Function

You have the option of using the Legacy Code Tool to generate a masked S-function block (graphical representation) that is configured to call a generated C MEX S-function. To generate such a block, call `legacy_code` with `'slblock_generate'` as the first argument and the name of the Legacy Code Tool data structure as the second argument.

```
legacy_code('slblock_generate', lct_spec);
```

The tool masks the block such that it displays the value of the `OutputFcnSpec` field. You can then add the block to a model manually.

If you prefer that the Legacy Code Tool add the block to a model automatically, specify the name of the model as a third argument. For example:

```
legacy_code('slblock_generate', lct_spec, 'myModel');
```

If the specified model (for example, `myModel.mdl`) exists, `legacy_code` opens the model and adds the masked S-function block described by the Legacy Code Tool data structure. If the model does not exist, the function creates a new model with the specified name and adds the masked S-function block.

Forcing Simulink Accelerator Mode to Use S-Function TLC Inlining Code

If you are using Simulink Accelerator mode, you can generate and force the use of TLC inlining code for the S-function generated by the Legacy Code Tool. To do this:

1 Generate a TLC block file for the S-function by calling the `legacy_code` function with `'sfcn_tlc_generate'` as the first argument and the name of the Legacy Code Tool data structure as the second argument.

```
legacy_code('sfcn_tlc_generate', lct_spec);
```

Consider the example in "Example of Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" on page 4-58. To generate a TLC file for the model shown at the end of that example, enter the following command:

```
legacy_code('sfcn_tlc_generate', def);
```

2 Force Accelerator mode to use the TLC file by using the `ssSetOptions` `SimStruct` function to set the S-function option `SS_OPTION_USE_TLC_WITH_ACCELERATOR`.

Calling Legacy C++ Functions

To call a legacy C++ function after initializing the Legacy Code Tool data structure, assign the value 'C++' to the `Options.language` field. For example,

```
def = legacy_code('initialize');
def.Options.language = 'C++';
```

To verify the new setting, enter

```
def.Options.language
```

Note The Legacy Code Tool can interface with C++ functions, but not C++ objects. For a work around, see “Legacy Code Tool Limitations” on page 4-75 in the Simulink documentation.

Handling Multiple Registration Files

You can have multiple registration files in the same directory and generate an S-function for each file with a single call to `legacy_code`. Likewise, you can use a single call to `legacy_code` to compile and link the S-functions and another to generate corresponding TLC block files, if appropriate.

Consider the following example, where `lct_register_1`, `lct_register_2`, and `lct_register_3` each create and initialize fields of a Legacy Code Tool structure.

```
defs1 = lct_register_1;
defs2 = lct_register_2;
defs3 = lct_register_3;
defs = [desfs1(:);defs2(:);defs3(:)];
```

You can then use the following sequence of calls to `legacy_code` to generate files based on the three registration files:

```
legacy_code('sfcn_cmex_generate', defs);
legacy_code('compile', defs);
legacy_code('sfcn_tlc_generate', defs);
```

Alternatively, you can process each registration file separately. For example:

```
defs1 = lct_register1;
legacy_code('sfcn_cmex_generate', defs1);
legacy_code('compile', defs1);
legacy_code('sfcn_tlc_generate', defs1);
.
.
.
defs2 = lct_register2;
legacy_code('sfcn_cmex_generate', defs2);
legacy_code('compile', defs2);
legacy_code('sfcn_tlc_generate', defs2);
.
.
.
defs3 = lct_register3;
legacy_code('sfcn_cmex_generate', defs3);
legacy_code('compile', defs3);
legacy_code('sfcn_tlc_generate', defs3);
```

Deploying Generated S-Functions

You can deploy the S-functions that you generate with the Legacy Code Tool for use by others. To deploy an S-function for simulation use only, you need to share only the compiled dynamically loadable executable.

Legacy Code Tool Demos

The Simulink product provides a set of demos that show applications of the Legacy Code Tool. To review the list of the demos, enter the following command in MATLAB Command Window and review the demos listed under the heading "Calling Legacy C and C++ Functions."

```
demo('simulink', 'modeling features')
```

Legacy Code Tool Limitations

Legacy Code Tool

- Generates C MEX S-functions for existing functions written in C or C++. The tool does not support transformation of MATLAB or Fortran functions.
- Can interface with C++ functions, but not C++ objects. One way of working around this limitation is to use the S-Function Builder to generate the shell of an S-function and then call the legacy C++ code from the S-function's `mdlOutputs` callback function.
- Does not support simulating continuous or discrete states. This prevents you from using the `mdlUpdate` and `mdlDerivatives` callback functions. If your application requires this support, see "Using the S-Function Builder to Incorporate Legacy Code" on page 2-17 in the Simulink S-function documentation.

- Always sets the S-functions flag for direct feedthrough (`sizes.DirFeedthrough`) to `true`. Due to this setting and the preceding limitation, the generated S-function cannot break algebraic loops.
- Supports only the continuous, but fixed in minor time step, sample time and offset option.
- Supports complex numbers, but only with Simulink built-in data types.
- Does not support use of function pointers as the output of the legacy function being called.
- Does not support the following S-function features:
  - Work vectors, other then general DWork vectors
  - Frame-based input and output signals
  - Port-based sample times
  - Multiple block-based sample times

How the Simulink Engine Interacts with C S-Functions

In this section...

Introduction

This section examines how the Simulink engine interacts with S-functions from two perspectives:

- **Process perspective**, i.e., at which points in a simulation the engine invokes the S-function.
- **Data perspective**, i.e., how the engine and the S-function exchange information during a simulation.

Process View

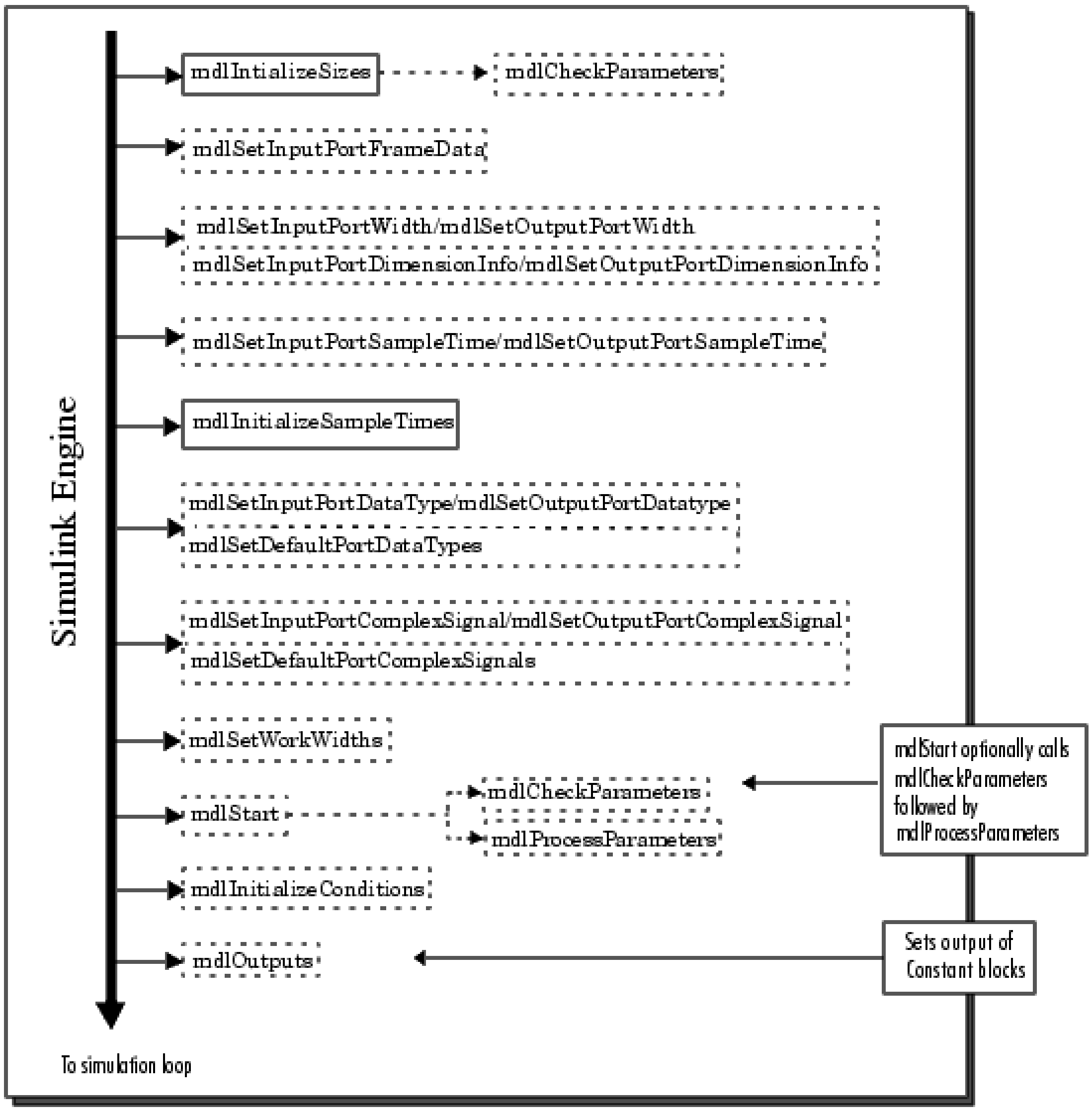
The following figures show the order in which the Simulink engine invokes the callback methods in an S-function. Solid rectangles indicate callbacks that always occur during model initialization or at every time step. Dotted rectangles indicate callbacks that may occur during initialization and/or at some or all time steps during the simulation loop. See the documentation for each callback method in Chapter 9, "S-Function Callback Methods — Alphabetical List" to determine the exact circumstances under which the engine invokes the callback.

Note The process view diagram represents the execution of S-functions that contain continuous and discrete states, enable zero-crossing detection, and reside in a model that uses a variable-step solver. Different solvers omit certain steps in the diagram. For a better understanding of how the Simulink engine executes your particular S-function, run the model containing the S-function using the Simulink debugger. See "Simulink Debugger" in *Using Simulink* for more information.

In the following model initialization loop, the Simulink engine configures the S-function for an upcoming simulation. The engine always makes the required calls to `mdlInitializeSizes` and `mdlInitializeSampleTime` to set up the fundamental attributes of the S-function, including input and output ports, S-function dialog parameters, work vectors, sample times, etc.

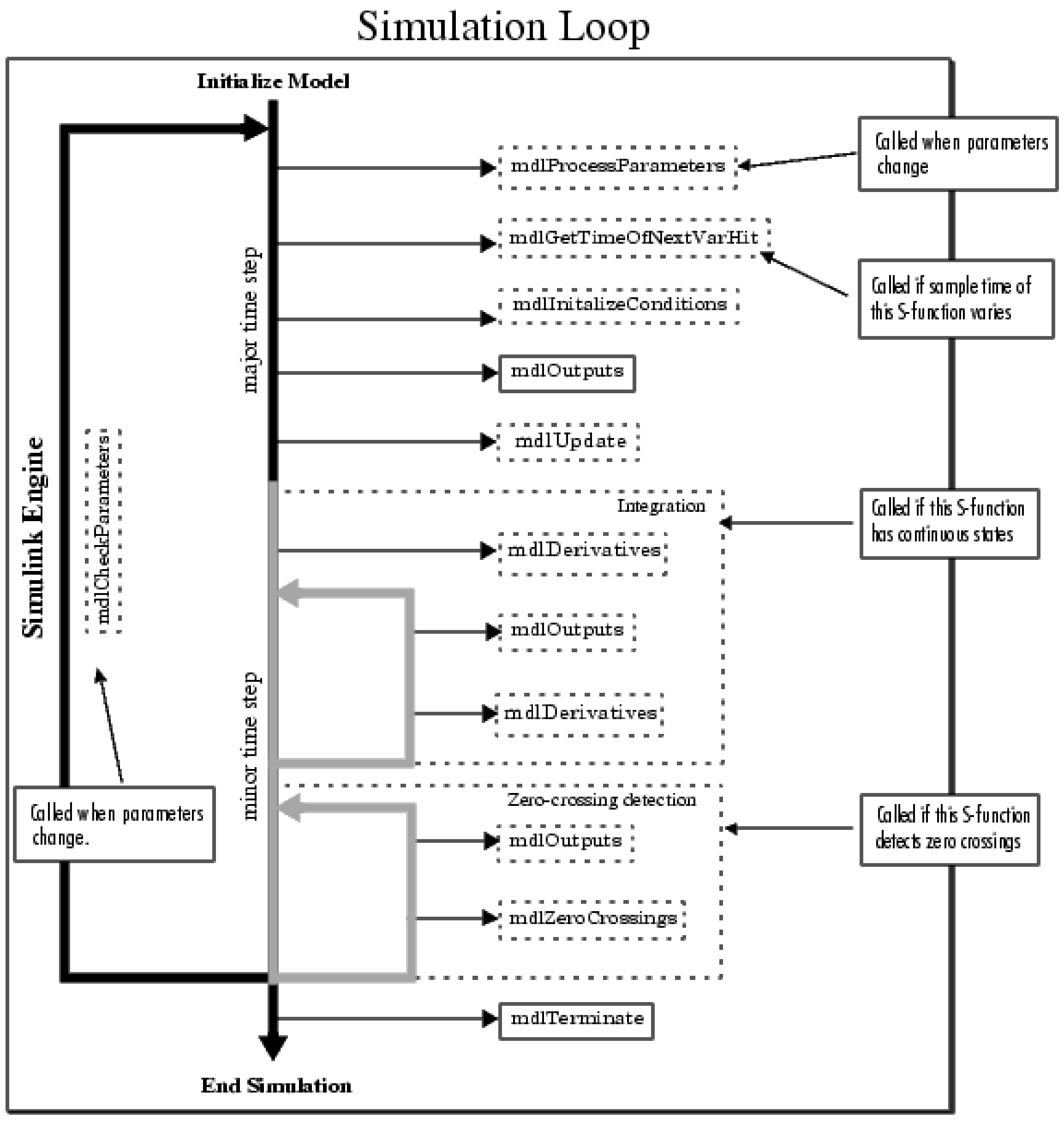
The engine calls additional methods, as needed, to complete the S-function initialization. For example, if the S-function uses work vectors, the engine calls `mdlSetWorkWidths`. Also, if the `mdlInitializeSizes` method deferred setting up input and output port attributes, the engine calls any methods necessary to complete the port initialization, such as `mdlSetInputPortWidth`, during signal propagation. The `mdlStart` method calls the `mdlCheckParameters` and `mdlProcessParameters` methods if the S-function uses dialog parameters.

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

Note The `mdlInitializeSizes` callback method also runs when you enter the name of a compiled S-function into the S-Function Block Parameters dialog box.

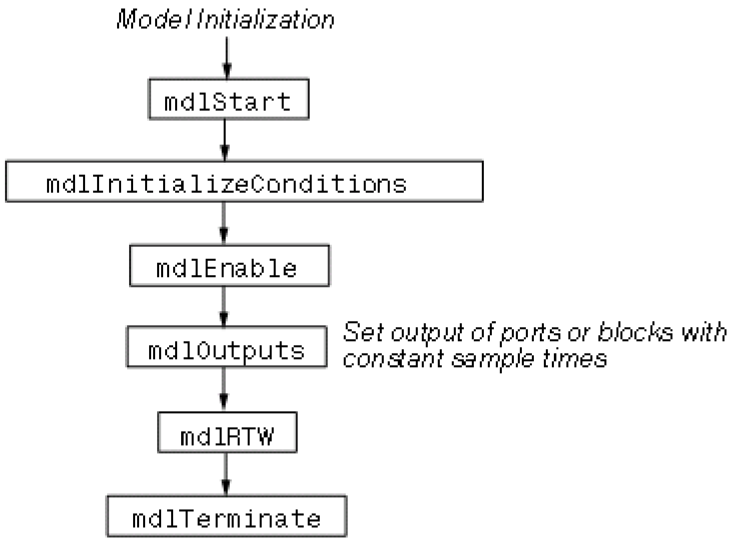
After initialization, the Simulink engine executes the following simulation loop. If the simulation loop is interrupted, either manually or when an error occurs, the engine jumps directly to the `mdlTerminate` method. If the simulation was manually halted, the engine first completes the current time step before invoking `mdlTerminate`.


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

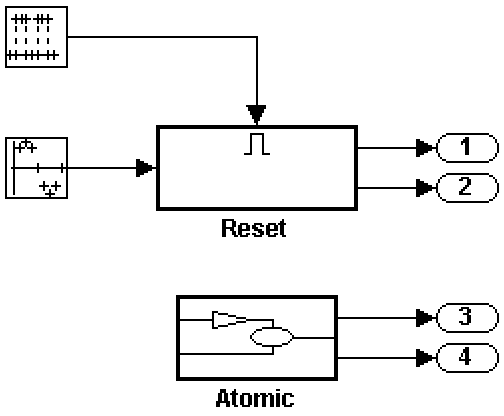
If your model contains multiple S-Function blocks, the engine invokes a particular methods for every S-function before proceeding to the next method. For example, the engine calls all the `mdlInitializeSizes` methods before calling any `mdlInitializeSampleTimes` methods. The engine uses the block sorted order to determine the order to execute the S-functions. See "What Is Sorted Order?" in *Using Simulink* to learn more about how the engine determines the block sorted order.

Calling Structure for Code Generation

If you use the Real-Time Workshop product to generate code for a model containing S-functions, the Simulink engine does not execute the entire calling sequence outlined above. Initialization proceeds as outlined above until the engine reaches the `mdlStart` method. The engine then calls the S-function methods show in the following figure, where the `mdlRTW` method is unique to the Real-Time Workshop product.



If the S-function resides in a conditionally executed subsystems, it is possible for the generated code to interleave calls to `mdlInitializeConditions` and `mdlStart`. Consider the following Simulink model `sfcndemo_enablesub.mdl`.



The model contains two nonvirtual subsystems, the conditionally executed enabled subsystem named Reset and the atomic subsystem named Atomic. Each subsystem contains an S-Function block that calls the S-function `dsfunc.c`, which models a discrete state-space system with two states. The enabled subsystem Reset resets the state values when the subsystem is enabled, and the output values when the subsystem is disabled.

Using the generic real-time (GRT) target, the generated code for the model-wide `Start` function calls the `Start` functions of the two subsystems before calling the model-wide `MdlInitialize` function, as shown in the following code:

```
void MdlStart(void)
{
  /* snip */

  /* Start for enabled SubSystem: '<Root>/Reset' */
  sfcndemo_enablesub_Reset_Start();

  /* end of Start for SubSystem: '<Root>/Reset' */

  /* Start for atomic SubSystem: '<Root>/Atomic' */
  sfcndemo_enablesub_Atomic_Start();
```

```
    /* end of Start for SubSystem: '<Root>/Atomic' */

  MdlInitialize();
```

The `Start` function for the enabled subsystem calls the subsystem's `InitializeConditions` function:

```
  void sfcndemo_enablesub_Reset_Start(void)
  {
     sfcndemo_enablesub_Reset_Init();
     /* snip */
  }
```

The `MdlInitialize` function, called in `MdlStart`, contains a call to the `InitializeConditions` function for the atomic subsystem:

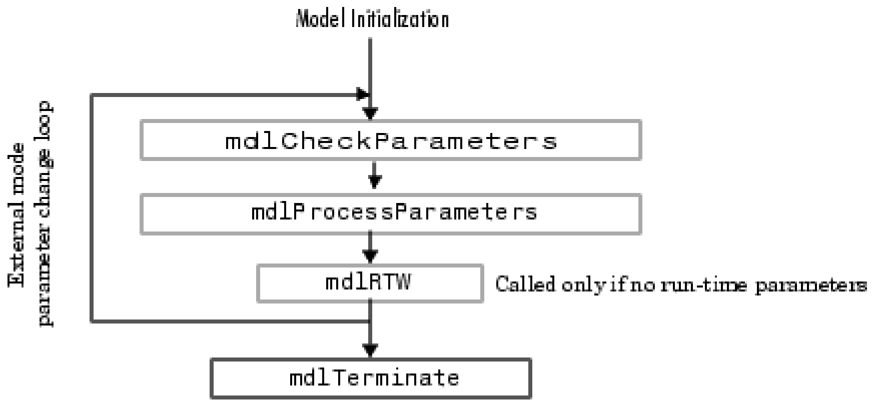
```
  void MdlInitialize(void)
  {
     /* InitializeConditions for atomic SubSystem:
          '<Root>/Atomic' */

     sfcndemo_enablesub_Atomic_Init();
  }
```

Therefore, the model-wide `Start` function interleaves calls to the `Start` and `InitializeConditions` functions for the two subsystems and the S-functions they contain.

For more information about the Real-Time Workshop product and how it interacts with S-functions, see "Integrating External Code With Generated C and C++ Code" in the *Real-Time Workshop User's Guide* and the *Real-Time Workshop Target Language Compiler* documentation.

Alternate Calling Structure for External Mode

When you are running a Simulink model in external mode, the calling sequence for S-function routines changes as shown in the following figure.



The engine 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 a Simulink model in external mode requires the Real-Time Workshop product. For more information about external mode, see the Real-Time Workshop documentation.

Data View

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

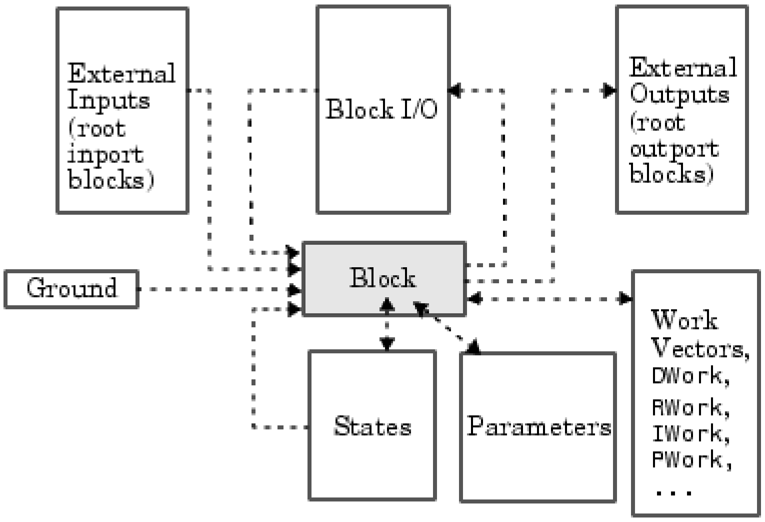
- External inputs via the root Inport blocks
- Ground if the input signal is unconnected or grounded

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

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

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

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


External Inputs (root inport blocks)
Block I/O
External Outputs (root outport blocks)
Ground
Block
Work Vectors, DWork, RWork, IWork, PWork, ...
States
Parameters

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

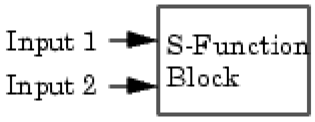
During the simulation loop, access the input signals using

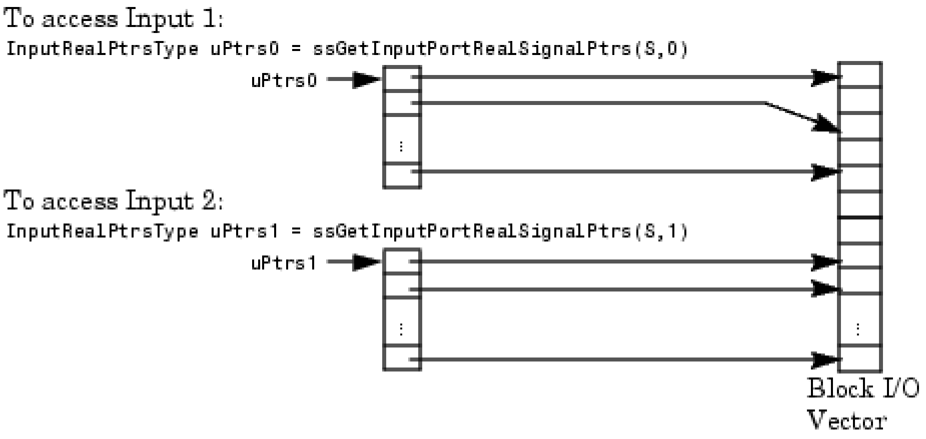
```
InputRealPtrsType uPtrs =
  ssGetInputPortRealSignalPtrs(S,portIndex)
```

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

```
*uPtrs[element]
```

The following figure describes how to access the input signals of an S-function with two inputs.




To access Input 1:
InputRealPtrsType uPtrs0 = ssGetInputPortRealSignalPtrs(S,0)
uPtrs0
To access Input 2:
InputRealPtrsType uPtrs1 = ssGetInputPortRealSignalPtrs(S,1)
uPtrs1
Block I/O Vector

As shown in the previous figure, the 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

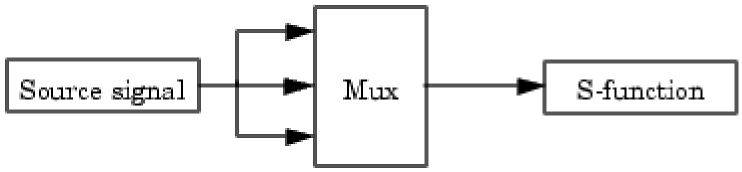
```
real_T *u = *uPtrs; /* Incorrect */
```

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 the Simulink software. 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 your S-function correctly accesses wide input signals, pass a replicated signal to each input port of your S-function. To do this, create a Mux block with the number of input ports equal to the width of the desired signal entering your S-function. Then, connect the driving source to each S-function input port, as shown in the following figure. Finally, run your S-function using this input signal to verify that it does not crash and produces expected results.



Writing Callback Methods

Writing an S-function basically involves creating implementations of the callback functions that the Simulink engine invokes during a simulation. For guidelines on implementing a particular callback, see the documentation for the callback in Chapter 9, “S-Function Callback Methods — Alphabetical List”. For information on using callbacks to implement specific block features, such as parameters or sample times, see Chapter 8, “Implementing Block Features”.

Using S-Functions in Normal Mode Referenced Models

When a C S-function appears in a referenced model that executes in Normal mode, successful execution is impossible if all of the following are true:

- The S-function has both an `mdlProcessParameters` function and an `mdlStart` function.
- The `mdlProcessParameters` function depends on the `mdlStart` function.
- The referenced model calls `mdlProcessParameters` before calling `mdlStart`.

Execution fails because `mdlProcessParameters` has dependency requirements that `mdlStart` has not satisfied. Automated analysis cannot guard against all possible causes of such failure: you must check your code manually and verify that `mdlProcessParameters` is not in any way dependent on `mdlStart` being called first. Examples of such dependency include:

- Allocating memory in `mdlStart` and using that memory in `mdlProcessParameters`. This is often done using `ssSetUserData` and `ssGetUserData`.
- Initializing any DWork or any global memory in `mdlStart` and reading the values in `mdlProcessParameters`.

To remind you to check for any such dependency problems, an error message appears by default for any S-function that is used in a Normal mode referenced model and contains both an `mdlProcessParameters` function and an `mdlStart` function. The error message does not mean that any dependency problems exist, but only that they might exist.

If you get such an error message, check for any problematic dependencies in the S-function, and recode as needed to eliminate them. When no such dependencies exist, you can safely suppress the error message and use the S-function in a Normal mode referenced model. To certify that the S-function is compliant, and the message is therefore unnecessary, include the following statement in `mdlInitializeSizes`:

```
ssSetModelReferenceNormalModeSupport (S, MDL_START_AND_MDL_PROCESS_PARAMS_OK);
```

For information about referenced models, see “Referencing a Model” in *Using Simulink*, and “Creating Model Components” in the Real-Time Workshop documentation.

Debugging C MEX S-Functions

In this section...

About Debugging C MEX S-Functions

This section provides high-level tips on how to debug C MEX S-functions within the Simulink environment and using third-party software. The following lists highlight some of the more common errors made when writing an S-function. For a more detailed analysis, use the debugger provided with your C compiler.

The examples at the end of this section show how to debug a C MEX S-function during simulation, using third-party software.

- The first example uses the Microsoft® Visual C++® .NET (version 7.0) environment.
- The second example debugs an S-function on The Open Group UNIX® platform.

Refer to your compiler documentation for further information on debugging files.

Debugging C MEX S-Functions in the Simulink Environment

Before you begin, make sure you have a good understanding of how to write C S-functions and the required callback methods. For assistance:

- Read the section Chapter 2, "Selecting an S-Function Implementation" to determine if you implemented your S-function using the most appropriate method.

- Use the S-Function Builder block to generate simple S-functions and study the contents of the source files.
- Inspect the S-function demo models available in `sfundemos.mdl`. The directory *`matlabroot`*`/simulink/src` contains the S-function source files for these models.

If your S-function is not compiling, first ensure that the `mex` command is properly configured and your S-function includes all necessary files:

- Run `mex -setup` to ensure that your compiler is correctly installed.
- Confirm that you are passing all the source files needed by your S-function to the `mex` command.
- Check that these additional source files are on the MATLAB path.
- Make sure that your S-function includes the `simstruc.h` header file. If you are accessing legacy code, make sure that any header files needed by that code are also included in your S-function.
- Make sure that your S-function does not include the `simstruc_types.h` or `rtwtypes.h` header files. These Simulink and Real-Time Workshop header files are automatically included for you. If you are compiling your S-function as a MEX-file for simulation, including the `rtwtypes.h` file results in errors.

If the `mex` command compiles your S-function, but your S-function does not simulate or the simulation produces incorrect results, inspect your S-function's source code to ensure that:

- You are not overwriting important memory
- You are not using any uninitialized variables

The following table describes additional common S-function constructs that can lead to compilation and simulation errors.

Does your S-function...	**Look for...**
Use `for` loops to assign memory?	Instances where your S-function might inadvertently assign values outside of the array bounds.

Does your S-function...	**Look for...**
Use global variables?	Locations in the code where the global variables can be corrupted. If you have multiple instances of your S-function in a model, they can write over the same memory location.
Allocate memory?	Memory your S-function does not deallocate. Always free memory that your S-function allocates, using the `malloc` and `free` commands to allocate and deallocate memory, respectively.

Have direct feedthrough?	An incorrect direct feedthrough flag setting in your S-function. An S-function can access its inputs in the `mdlOutputs` method only if it specifies that the input ports have direct feedthrough. Accessing input signals in `mdlOutputs` when the input port direct feedthrough flag is set to `false` leads to indeterminate behavior. To check if you have a direct feedthrough flag incorrectly set, you can turn on the model property `TryForcingSFcnDF` using the command
	
	`set_param(`*`model_name`*`,'TryForcingSFcnDF','on')`
	
	This command specifies that all S-functions in the model *`model_name`* have a direct feedthrough flag of `true` for all their input ports. After you turn on this property, if your simulation produces correct answers without causing an algebraic loop, one of your S-functions in the model potentially set an incorrect direct feedthrough flag. Consult the file
	
	`matlabroot/simulink/src/sfuntmpl_directfeed.txt`
	
	for more information on diagnosing direct feedthrough errors.

Does your S-function...	Look for...

Access input signals correctly?	Instances in the code where your S-function uses incorrect macros to access input signals, for example when accessing a discontiguous signal. Discontiguous signals result when an S-function input port is fed by a Selector block that selects every other element of a vector signal. For discontiguous input signals, use the following commands:
	`// In mdlInitializeSizes`
	`ssSetInputPortRequiredContiguous(S, 0, 0);`
	
	`// In mdlOutputs, access the inputs using`
	`InputRealPtrsType uPtrs1 =`
	`    ssGetInputPortRealSignalPtrs(S,0);`
	
	For contiguous input signals, use the following commands:
	`// In mdlInitializeSizes`
	`ssSetInputPortRequiredContiguous(S, 0, 1);`
	
	`// In mdlOutputs, access the inputs using`
	`const real_T  *u0  =`
	`   (const real_T*) ssGetInputPortSignal(S,0);`
	
	`/* If ssSetInputPortRequiredContiguous is 0,`
	`ssGetInputPortSignal returns an invalid pointer.*/`

Debugging Techniques

You can use the following techniques for additional assistance with debugging your S-function.

- Compile the S-function in debug mode using the `-g` option for the `mex` command. This enables additional diagnostics features that are called only when you compile your S-function in debug mode.

- Place `ssPrintf` statements inside your callback methods to ensure that they are running and that they are executing in the order you expect. Also, use `ssPrintf` statements to print return values to the MATLAB command prompt to check if your code is producing the expected results.
- Type `feature memstats` at the MATLAB command prompt to query the memory usage.
- Use the MATLAB File & Directory Comparisons tool, or other text differencing application, to look for textual changes in different versions of your S-function. This can help you locate changes that disabled an S-function that previously compiled and ran. See "Comparing Files and Folders" in *MATLAB Desktop Tools and Development Environment* for instructions on how to use the File & Directory Comparisons tool.
- Use settings on the Configuration Parameters dialog box to check for memory problems.
  - Set the **Solver data inconsistency** diagnostic to `warning`.
  - Set the **Array bounds exceeded** diagnostic to `warning` or `error.` (See "Checking Array Bounds" on page 8-72 for more information on how to use this diagnostic.)
  - Turn the **Signal storage reuse** optimization off.
- Separate the S-function's algorithm from its Simulink interface then use the S-Function Builder to generate a new Simulink interface for the algorithm. The S-Function Builder ensures that the interface is implemented in the most consistent method.

Debugging C MEX S-Functions Using Third-Party Software

You can debug and profile the algorithm portion of your S-function using third-party software if you separate the algorithm from the S-function's Simulink interface. You cannot debug and profile the S-function's interface with the Simulink engine because the Simulink interface code does not ship with the product.

You can additionally use third-party software to debug an S-function during simulation, as shown in the following two examples. These examples use the Simulink model `sfcndemo_timestwo.mdl` and the C MEX S-function `timestwo.c`.

Debugging C MEX S-Functions Using the Microsoft Visual C++ .NET Environment

Before beginning the example, save the files `sfcndemo_timestwo.mdl` and `timestwo.c` into your working directory.

1. Open the Simulink model `sfcndemo_timestwo.mdl`.

2. Create a debuggable version of the MEX-file by compiling the C-file using the `mex` command with the `-g` option.

   ```
   mex -g timestwo.c
   ```

   The `-g` option creates the executable `timestwo.mexw32` with debugging symbols included. At this point, you may want to simulate the `sfcndemo_timestwo` model to ensure it runs properly.

3. Without exiting the MATLAB environment, start the Microsoft Development Environment.

4. From the Microsoft Development Environment menu bar, select **Tools > Debug Processes**.

5. In the **Processes** dialog box that opens:

   a. Select the `MATLAB.exe` process in the **Available Processes** list.

   b. Click **Attach**.

6. In the **Attach to Process** dialog box that opens:

   a. Select **Native** in the list of program types to debug.

   b. Click **OK**.
   You should now be attached to the MATLAB process.

7. Click **Close** on the **Processes** dialog box.

8. From the Microsoft Development Environment **File** menu, select **Open > File**. Select the `timestwo.c` source files from the file browser that opens.

9. Set a breakpoint on the desired line of code by right-clicking on the line and selecting **Insert Breakpoint** from the context menu. If you have not previously run the model, the breakpoint may show up with a question

mark, indicating that the executable is not loaded. Subsequently running the model loads the `.mexw32` file and removes the question mark from the breakpoint.

10 Start the simulation from the `sfcndemo_timestwo` Simulink model. You should be running the S-function in the Microsoft Development Environment and can debug the file within that environment.

Debugging C MEX S-Functions on The Open Group UNIX Platforms

Before beginning the example, save the files `sfcndemo_timestwo.mdl` and `timestwo.c` into your working directory.

1 Open the Simulink model `sfcndemo_timestwo.mdl`.

2 Create a debuggable version of the MEX-file by compiling the C-file using the `mex` command with the `-g` option

```
mex -g timestwo.c
```

The `-g` option creates the executable `timestwo.mexw32` with debugging symbols included. At this point, you may want to simulate the `sfcndemo_timestwo` model to ensure it runs properly.

3 Exit the MATLAB environment.

4 Start the MATLAB environment in debugging mode using the following command.

```
matlab -D<nameOfDebugger>
```

The `-D` flag starts the MATLAB environment within the specified debugger. For example, to use the `dbx` debugging tool on Sun™ Solaris™ platform, enter the following command.

```
matlab -Ddbx
```

5 Once the debugger has loaded, continue loading the MATLAB environment by typing `run` at the debugger prompt.

```
(dbx) run
```

```
Running: matlab
(process id 9375)
```

6 After the MATLAB environment starts, enable debugging by entering the following command at the MATLAB command prompt.

```
dbmex on
```

7 Open the `sfcndemo_timestwo` Simulink model.

8 Simulate the model. This brings you into the debugger.

9 Set breakpoints in the source code, for example:

```
(dbx) stop in mdlOutputs
(2) stop in `timestwo.mexs64`timestwo.c`mdlOutputs
(dbx) file timestwo.c
```

10 Issue the `cont` command to continue.

```
(dbx) cont
```

11 Use your debugger routines to debug the S-function.

Converting Level-1 C MEX S-Functions to Level-2

In this section...

Guidelines for Converting Level-1 C MEX S-Functions to Level-2

Level-2 S-functions were introduced with Simulink version 2.2. Level-1 S-functions refer to S-functions that were written to work with Simulink version 2.1 and previous releases. Level-1 S-functions are compatible with Simulink version 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)
  {
  }
  ```

  Otherwise, delete the function.

- Replace all obsolete `SimStruct` macros. See "Obsolete Macros" on page 4-104 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[1] 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
  ```

1. UNIX is a registered trademark of The Open Group in the United States and other countries.

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. Replace each obsolete macro with the macro specified in the following table.

Obsolete Macro	Replace with
`ssGetU(S), ssGetUPtrs(S)`	`ssGetInputPortSignalPtrs(S,port)`, `ssGetInputPortSignal(S,port)`
`ssGetY(S)`	`ssGetOutputPortRealSignal(S,port)`
`ssGetX(S)`	`ssGetContStates(S), ssGetRealDiscStates(S)`
`ssGetStatus(S)`	Normally not used, but `ssGetErrorStatus(S)` is available.
`ssSetStatus(S,msg)`	`ssSetErrorStatus(S,msg)`
`ssGetSizes(S)`	Specific call for the wanted item (i.e., `ssGetNumContStates(S)`)
`ssGetMinStepSize(S)`	No longer supported.
`ssGetPresentTimeEvent(S,sti)`	`ssGetTaskTime(S,`*`sti`*`)`
`ssGetSampleTimeEvent(S,sti)`	`ssGetSampleTime(S,`*`sti`*`)`
`ssSetSampleTimeEvent(S,t)`	`ssSetSampleTime(S,`*`sti`*`,`*`t`*`)`
`ssGetOffsetTimeEvent(S,sti)`	`ssGetOffsetTime(S,`*`sti`*`)`
`ssSetOffsetTimeEvent(S,sti,t)`	`ssSetOffsetTime(S,`*`sti`*`,`*`t`*`)`
`ssIsSampleHitEvent(S,sti,tid)`	`ssIsSampleHit(S,`*`sti`*`,`*`tid`*`)`
`ssGetNumInputArgs(S)`	`ssGetNumSFcnParams(S)`
`ssSetNumInputArgs(S, numInputArgs)`	`ssSetNumSFcnParams(S,`*`numInputArgs`*`)`
`ssGetNumArgs(S)`	`ssGetSFcnParamsCount(S)`

Obsolete Macro	Replace with
`ssGetArg(S,argNum)`	`ssGetSFcnParam(S,`*`argNum`*`)`
`ssGetNumInputs`	`ssGetNumInputPorts(S)` and `ssGetInputPortWidth(S,`*`port`*`)`
`ssSetNumInputs`	`ssSetNumInputPorts(S,`*`nInputPorts`*`)` and `ssSetInputPortWidth(S,`*`port`*`,`*`val`*`)`
`ssGetNumOutputs`	`ssGetNumOutputPorts(S)` and `ssGetOutputPortWidth(S,`*`port`*`)`
`ssSetNumOutputs`	`ssSetNumOutputPorts(S,`*`nOutputPorts`*`)` and `ssSetOutputPortWidth(S,`*`port`*`,`*`val`*`)`

Creating C++ S-Functions

The procedure for creating C++ S-functions is nearly the same as that for creating C S-functions (see Chapter 4, “Writing S-Functions in C”). The following sections explain the differences.

Creating a C++ Source File

To create a C++ S-function from a C S-function refer to a C++ reference. In addition, set up the MEX function to use a C++ compiler (see “Building MEX-Files”)

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
    y[0] = c->output();                          // use 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 `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_cpp.cpp` file, enter

```
mex sfun_counter_cpp.cpp
```

at the MATLAB command prompt.

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

Creating Fortran S-Functions

Level-1 Versus Level-2 S-Functions

There are two main strategies to executing Fortran code from the Simulink software. 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 product. To have complete code generation with the Real-Time Workshop product, you must inline the Fortran S-function. See “Inlining S-Functions” in the Target Language Compiler documentation for more information.

The original S-function interface was called the Level-1 API. As the Simulink product 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).

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

Creating Level-1 Fortran S-Functions

In this section...

Fortran MEX Template File

A template file for Fortran MEX S-functions is located at *matlabroot*/simulink/src/sfuntmpl_fortran.F. 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 of a Level-1 Fortran S-Function

The example file, *matlabroot*/simulink/src/sfun_timestwo_for.F, 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.F simulink.F
C
```

```
C     Copyright 1990-2002 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 command prompt and run the model.

Inline Code Generation Example

Real-Time Workshop users can use the sample block target file `sfun_timestwo_for.tlc` to generate inlined code for `sfcndemo_timestwo_for.mdl`. If you want to learn how to inline your own Fortran MEX-file, see "Inlining S-Functions" in the Real-Time Workshop Target Language Compiler documentation.

Creating Level-2 Fortran S-Functions

In this section...

About 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 the Simulink software 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 you have legacy Fortran code that exists as a standalone simulation, there is some work to be done to identify parts of the code that need to be registered with the Simulink software, 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 6-18).

Template File

The file `sfuntmpl_gate_fortran.c` contains a template for creating a C MEX-file S-function that invokes a Fortran subroutine in its `mdlOutputs` method. It works with a simple Fortran subroutine if you modify the Fortran subroutine name in the code. The template allocates DWork vectors to store the data that communicates with the Fortran subroutine. See Chapter 7, "Using Work Vectors" for information on setting up DWork vectors.

C/Fortran Interfacing Tips

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

MEX Environment

Remember that `mex -setup` needs to find both the MATLAB, C, and the Fortran compilers, but it can work with only one of these compilers at a time. If you install or change compilers, you must run `mex -setup` between other `mex` commands.

Test the installation and setup using sample MEX-files from the MATLAB, C, and Fortran MEX examples in *`matlabroot`*`/extern/examples/mex`, as well as S-function examples.

If using a C compiler on a Microsoft Windows platform, test the `mex` setup using the following commands and the example C source code file, `yprime.c`, in *`matlabroot`*`\extern\examples\mex`.

```
cd([matlabroot '\extern\examples\mex'])
mex yprime.c
```

If using a Fortran compiler, test the `mex` setup using the following commands and the example Fortran source code files, `yprime.F` and `yprimefg.F`, in *`matlabroot`*`\extern\examples\mex`.

```
cd([matlabroot '\extern\examples\mex'])
mex yprimef.f yprimefg.f
```

For more information, see Building MEX-Files in the *MATLAB External Interfaces* documentation.

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. **Intel**® Visual Fortran (the replacement for Compaq® Visual Fortran) has the default stack cleanup as the caller.

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 the Windows platform as well. The MKS, Inc. company provides commercial versions of powerful utilities for The Open Group UNIX platforms. Additional utilities 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 a Linus Torvalds' Linux® platform.

```
0000115 E€
0000136 E€
0000271 E€
0000467 ˙E€@
0000530 ˙E€
0000575 E€ E 5@
0001267 Cf VC- :C
0001323 :|.-:8˘#8 Kw6
0001353 ?333@
0001364 333
0001414 01.01
0001425 GCC: (GNU) egcs-2.91.66 19990314/
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 Visual Fortran on 32-bit Windows machines, 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 a UNIX system, 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 the 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 32-bit Windows machines, using Microsoft Visual C++ and Intel Visual Fortran 10.1, this example can be compiled using the following two `mex` commands. Enter each command on one line. The `mex -setup` command must be run to return to the C compiler before executing the second command. In the second command, replace the variable `IFORT_COMPILER10` with the name of the system's environment variable pointing to the Visual Fortran 10.1 root directory on your system.

```
mex -v  -c fullfile(matlabroot,'simulink','src','sfun_atmos_sub.F'),
-f fullfile(matlabroot,'bin','win32','mexopts','intelf10msvs2005opts.bat'))

!mex -v -L"%IFORT_COMPILER10%\IA32\LIB" -llibifcoremd -lifconsol
-lifportmd -llibmmd -llibirc sfun_atmos.c sfun_atmos_sub.obj
```

On 64-bit Windows machines, using Visual C++® and Visual Fortran 10.1, this example can be compiled using the following two `mex` commands (each command is on one line). The `mex -setup` command must be run to return to the C compiler before executing the second command. The variable `IFORT_COMPILER10` is the name of the system's environment variable pointing to the Visual Fortran 10.1 root directory and may vary on your system. Replace *`matlabroot`* with the path name to your MATLAB root directory.

```
mex -v  -c fullfile(matlabroot,'simulink','src','sfun_atmos_sub.F'),
-f fullfile(matlabroot,'bin','win64','mexopts','intelf10msvs2005opts.bat'))

!mex -v -L"%IFORT_COMPILER10%\EM64T\LIB" -llibifcoremd -lifconsol
```

```
-lifportmd -llibmmd -llibirc sfun_atmos.c sfun_atmos_sub.obj
```

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.

Choosing a Fortran Compiler

On a Windows machine, using Visual C++ with Fortran is best done with Visual Fortran 10.1.

For an up-to-date list of all the supported compilers, see the MathWorks supported and compatible compiler list at:

```
http://www.mathworks.com/support/compilers/current_release/
```

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 the Simulink engine 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.

If instead the code needs to have continuous time states with support for variable-step solvers, the states must be registered and stored with the engine 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 the Simulink software 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 standalone simulation to work inside the Simulink environment, identify the main loop of time integration and remove the loop and, if you want the Simulink engine 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 S-function demo sfcndemo_atmos.mdl contains an example of a C MEX S-function calling a Fortran subroutine. The Fortran subroutine Atmos is in the file sfun_atmos_sub.F. This subroutine calculates the standard atmosphere up to 86 kilometers. The subroutine has four arguments.

```
SUBROUTINE Atmos(alt, sigma, delta, theta)
```

The gateway C MEX S-function, sfun_atmos.c, declares the Fortran subroutine.

```
/*
 * Windows uses upper case for Fortran external symbols
 */
#ifdef _WIN32
#define atmos_ ATMOS
#endif

extern void atmos_(float *alt,
                   float *sigma,
                   float *delta,
                   float *theta);
```

The mdlOutputs method calls the Fortran subroutine using pass-by-reference for the arguments.

```
    /* call the Fortran routine using pass-by-reference */
    atmos_(&falt, &fsigma, &fdelta, &ftheta);
```

To see this example working in the sample model sfcndemo_atmos.mdl, enter the following command at the MATLAB command prompt.

```
sfcndemo_atmos
```

Building Gateway C MEX S-Functions on a Windows System

On 32-bit Windows systems using Visual C++ and Visual Fortran 10.1, you need to use separate commands to compile the Fortran file and then link it to the C gateway file. Each command is on one line.

1. Run `mex -setup` and select a Fortran compiler.
2. Compile the Fortran file using the following command. Enter the command on one line.

   ```
   mex -v  -c fullfile(matlabroot,'simulink','src','sfun_atmos_sub.F'),
   -f fullfile(matlabroot,'bin','win32','mexopts','intelf10msvs2005opts.bat'))
   ```

3. Rerun `mex -setup` and select a C compiler.
4. Link the compiled Fortran subroutine to the gateway C MEX S-function using the following command. The variable `IFORT_COMPILER10` is the name of the system's environment variable pointing to the Visual Fortran 10.1 root directory and may vary on your system.

   ```
   !mex -v -L"%IFORT_COMPILER10%\IA32\LIB" -llibifcoremd
   -lifconsol -lifportmd -llibmmd -llibirc sfun_atmos.c
   sfun_atmos_sub.obj
   ```

Building Gateway C MEX S-Functions on a UNIX System

Build the gateway on a UNIX system using the command

```
mex sfun_atmos.c sfun_atmos_sub.o
```

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

If the `libf2c.a` library is not 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
```

Porting Legacy Code

In this section...

Find the States

If a variable-step solver is being used, it is critical that all continuous states are identified in the code and put into the C S-function'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 C 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 the Simulink engine, 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 major simulation 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. See "How to Use DWork Vectors" on page 7-7 for details on allocating data-typed work vectors.

Sample Times

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

Store Data

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 Chapter 7, "Using Work Vectors").

If you plan to have only one copy of the S-function in the model, DWork vectors still provide the most advanced method for storing data. However, another alternative is to allocate a block of memory using the `malloc` command and store the pointer to that memory in a PWork vector (see "Elementary Work Vectors" on page 7-29). In this case, you must remember to deallocate the memory using the `free` command in the S-function's `mdlTerminate` method.

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.

Using Work Vectors

About DWork Vectors

In this section...

What is a DWork Vector?

DWork vectors are blocks of memory that an S-function asks the Simulink engine to allocate to each instance of the S-function in a model. If multiple instances of your S-function can occur in a model, your S-function must use DWork vectors instead of global or static memory to store instance-specific values of S-function variables. Otherwise, your S-function runs the risk of one instance overwriting data needed by another instance, causing a simulation to fail or produce incorrect results. 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 DWork vectors that the engine manages for each particular instance of the S-function.

Advantages of DWork Vectors

DWork vectors have several advantages:

- Provide instance-specific storage for block variables
- Support floating-point, integer, pointer, and general data types
- Eliminate static and global variables
- Interact directly with the Simulink engine to perform memory allocation, initialization, and deallocation
- Facilitate inlining the S-function during code generation
- Provide more control over how data appears in the generated code

---

Note DWork vectors are the most generalized and versatile type of work vector and the following sections focus on their use. The Simulink product provides additional elementary types of work vectors that support floating-point, integer, pointer, and mode data. You can find a discussion of these work vectors in "Elementary Work Vectors" on page 7-29.

DWork vectors provide the most flexibility for setting data types, names, etc., of the data in the simulation and during code generation. The following list describes all the properties that you can set on a DWork vector:

- Data type
- Size
- Numeric type, either real or complex
- Name
- Usage type (see "Types of DWork Vectors" on page 7-5)
- Real-Time Workshop identifier
- Real-Time Workshop storage class
- Real-Time Workshop C type qualifier

See "How to Use DWork Vectors" on page 7-7 for instructions on how to set these properties. The three Real-Time Workshop properties pertain only to code generation and have no effect during simulation.

DWork Vectors and the Simulink Engine

A key advantage of DWork vectors is their connection to the Simulink engine. Over the course of the simulation, the engine relieves the S-function of all memory management tasks related to DWork vectors.

To see how this connection is useful, consider an S-function that uses a global variable to store data. If more than one copy of the S-function exists in a model, each instance of the S-function must carefully allocate, manipulate, and deallocate each piece of memory it uses.

In an S-function that uses DWork vectors, the engine, not the S-function, manages the memory for the DWork vector. At the start of a simulation, the engine allocates the memory required for each instance of the S-function based on the size and the data type of the DWork vector contents. At the end of the simulation, the engine automatically deallocates the memory.

Note You have no control over how the engine allocates memory for DWork vectors during simulation. When using the Real-Time Workshop software, you can use storage classes to customize the memory allocation during code generation. See the reference page for `ssSetDWorkRTWStorageClass` for more information on using storage classes.

The engine also performs special tasks based on the type of DWork vector used in the S-function. For example, it includes DWork vectors that store discrete state information in the model-wide state vector and makes them available during state logging.

DWork Vectors and the Real-Time Workshop Product

DWork vectors allow you to customize how data appears in the generated code. When code is generated, the Real-Time Workshop code generator includes the DWork vector in the data structure for the model. The DWork vector controls the field name used in the structure. DWork vectors also control the storage class and C type qualifier used in the generated code. See `sfun_rtwdwork.c` for an example.

Types of DWork Vectors

All DWork vectors are S-function memory that the Simulink engine manages. The Simulink software supports four types of DWork vectors:

- **General DWork vectors** contain information of any data type.
- **DState vectors** contain discrete state information. Information stored in a DState vector appears as a state in the linearized model and is available during state logging.
- **Scratch vectors** contain values that do not need to persist from one time step to the next.
- **Mode vectors** contain mode information, usually stored as Boolean or integer data.

S-functions register the DWork vector type using the `ssSetDWorkUsageType` macro. This macro accepts one of the four usage types described in the following table.

DWork	Usage Type	Functionality
General	`SS_DWORK_USED_AS_DWORK`	Store instance specific persistent data. General DWork vectors can also be used to store discrete state and mode data, however the Simulink engine will not treat this information specially. You might choose to use a general DWork vector to store state information if you want to avoid data logging.
DState	`SS_DWORK_USED_AS_DSTATE`	Store discrete state information. Using the DState vector instead of `ssSetNumDiscStates` to store discrete states provides more flexibility for naming and data typing the states. The engine marks blocks with discrete states as special during sample time propagation. In addition, the engine makes the data stored in the DState vector available during data logging.

DWork	Usage Type	Functionality
Mode	`SS_DWORK_USED_AS_MODE`	Indicate to the Simulink engine that the S-function contains modes. The engine handles blocks with modes specially when solving algebraic loops. In addition, the engine updates an S-function with modes only at major time steps. DWork mode vectors are more efficient than standard mode work vectors (see “Elementary Work Vectors” on page 7-29) because they can store mode information as Boolean data. In addition, while an S-function has only one mode work vectors, it can have multiple DWork vectors configured to store modes.
Scratch	`SS_DWORK_USED_AS_SCRATCH`	Store memory that is not persistent, for example, a large variable that you do not want to mark on the stack. Scratch vectors are scoped to a particular S-function method (for example, `mdlOutputs`) and exist across a single time step. Scratch memory can be shared across S-function blocks. The Simulink engine attempts to minimize the amount of memory used by scratch variables and reuses scratch memory whenever possible.

How to Use DWork Vectors

In this section...

Using DWork Vectors in C MEX S-Functions

The following steps show how to initialize and use DWork vectors in a C MEX S-function. For a full list of `SimStruct` macros pertaining to DWork vectors, see "DWork Vector C MEX Macros" on page 7-10.

1. In `mdlInitializeSizes`, specify the number of DWork vectors using the `ssSetNumDWork` macro. For example, to specify that the S-function contains two DWork vectors, use the command

   ```
   ssSetNumDWork(S, 2);
   ```

   Although the `mdlInitializeSizes` method tells the Simulink engine how many DWork vectors the S-function will use, the engine does not allocate memory for the DWork vectors, at this time.

   An S-function can defer specifying the number of DWork vectors until all information about the S-function's inputs is available by passing the value `DYNAMICALLY_SIZED` to the `ssSetNumDWork` macro. If an S-function defers specifying the number of DWork vectors in `mdlInitializeSizes`, it must provide a `mdlSetWorkWidths` method to set up the DWork vectors.

2. If the S-function does not provide an `mdlSetWorkWidths` method, the `mdlInitializeSizes` method sets any applicable attributes for each DWork vector. For example, the following lines initialize the widths and data types of the DWork vectors initialized in the previous step.

   ```
   ssSetDWorkWidth(S, 0, 2);
   ssSetDWorkWidths(S, 1, 1);
   ```

```
ssSetDWorkDataType(S, 0, SS_DOUBLE);
ssSetDWorkDataType(S, 1, SS_BOOLEAN);
```

The following table lists attributes you can set for a DWork vector and shows an example of the macro that sets it.

Attribute	Macro
Data type	`ssSetDWorkDataType(S, 0, SS_DOUBLE);`
Size	`ssSetDWorkWidth(S, 0, 2);`
Name	`ssSetDWorkName(S, 0, "sfcnState");`
Usage type	`ssSetDWorkUsageType(S, 0, SS_DWORK_USED_AS_DSTATE);`
Numeric type, either real or complex	`ssSetDWorkComplexSignal(S, 0, COMPLEX_NO);`
Real-Time Workshop identifier	`ssSetDWorkRTWIdentifier(S, 0, "Gain");`
Real-Time Workshop storage class	`ssSetDWorkRTWStorageClass(S, 0, 2);`
Real-Time Workshop C type qualifier	`ssSetDWorkRTWTypeQualifier(S, 0, "volatile");`

See `ssSetDWorkRTWStorageClass` for a list of supported storage classes.

3 In `mdlStart`, initialize the values of any DWork vectors that should be set only at the beginning of the simulation. Use the `ssGetDWork` macro to retrieve a pointer to each DWork vector and initialize the values. For example, the following `mdlStart` method initializes the first DWork vector.

```
static void mdlStart(SimStruct *S)
{
    real_T *x = (real_T*) ssGetDWork(S,0);

    /*  Initialize the first DWork vector */
    x[0] = 0;
    x[1] = 2;
}
```

The Simulink engine allocates memory for the DWork vector before calling the `mdlStart` method. Because the `mdlStart` method is called only once at the beginning of the simulation, do not use it for data or states that need to be reinitialized, for example, when reenabling a disabled subsystem containing the S-function.

4. In `mdlInitializeConditions`, initialize the values of any DWork vectors that need to be reinitialized at certain points in the simulation. The engine executes `mdlInitializeConditions` at the beginning of the simulation and any time an enabled subsystem containing the S-function is reenabled. See the `mdlStart` example in the previous step for the commands used to initialize DWork vector values.

5. In `mdlOutputs`, `mdlUpdate`, etc., use the `ssGetDWork` macro to retrieve a pointer to the DWork vector and use or update the DWork vector values. For example, for a DWork vector storing two discrete states, the following `mdlOutputs` and `mdlUpdate` methods calculate the output and update the discrete state values.

   The S-function previously defined `U(element)` as `(*uPtrs[element])` and `A`, `B`, `C`, and `D` as the state-space matrices for a discrete state-space system.

```
/* Function: mdlOutputs ==============================================
 * Abstract:
 *      y = Cx + Du
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    if( ssGetDWorkUsageType(S, 0) == SS_DWORK_USED_AS_DSTATE) {
        real_T            *y      = ssGetOutputPortRealSignal(S,0);
        real_T            *x      = (real_T*) ssGetDWork(S, 0);
        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       = (real_T*) ssGetDWork(S, 0);
    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];
}
```

You do not have to include any code in the `mdlTerminate` method to deallocate the memory used to store the DWork vector. Similarly, if you are generating inlined code for the S-function, you do not have to write an `mdlRTW` method to access the DWork vector in the TLC file. The Simulink software handles these aspects of the DWork vector for you.

DWork Vector C MEX Macros

The following table lists the C MEX macros pertaining to DWork vectors.

Macro	Description
`ssSetNumDWork`	Specify the number of DWork vectors.

Macro	Description
ssGetNumDWork	Query the number of DWork vectors.
ssGetDWork	Get a pointer to a specific DWork vector.
ssGetDWorkComplexSignal	Determine if a specific DWork vector is real or complex.
ssGetDWorkDataType	Get the data type of a DWork vector.
ssGetDWorkName	Get the name of a DWork vector.
ssGetDWorkRTWIdentifier	Get the identifier used to declare a DWork vector in the generated code.
ssGetDWorkRTWIdentifierMustResolveToSignalObject	Indicate if a DWork vector must resolve to a `Simulink.Signal` object in the MATLAB workspace.
ssGetDWorkRTWStorageClass	Get the storage class of a DWork vector.
ssGetDWorkRTWTypeQualifier	Get the C type qualifier used to declare a DWork vector in the generated code.
ssGetDWorkUsageType	Determine how a DWork vector is used in the S-function.
ssGetDWorkUsedAsDState	Determine if a DWork vector stores discrete states.
ssGetDWorkWidth	Get the size of a DWork vector.
ssSetDWorkComplexSignal	Specify if the elements of a DWork vector are real or complex.
ssSetDWorkDataType	Specify the data type of a DWork vector.
ssSetDWorkName	Specify the name of a DWork vector.

Macro	Description
`ssSetDWorkRTWIdentifier`	Specify the identifier used to declare a DWork vector in the generated code.
`ssSetDWorkRTWIdentifierMustResolveToSignalObject`	Specify if a DWork vector must resolve to a `Simulink.Signal` object.
`ssSetDWorkRTWStorageClass`	Specify the storage class for a DWork vector.
`ssSetDWorkRTWTypeQualifier`	Specify the C type qualifier used to declare a DWork vector in the generated code.
`ssSetDWorkUsageType`	Specify how a DWork vector is used in the S-function.
`ssSetDWorkUsedAsDState`	Specify that a DWork vector stores discrete state values.
`ssSetDWorkWidth`	Specify the width of a DWork vector.

Using DWork Vectors in Level-2 M-File S-Functions

The following steps show how to initialize and use DWork vectors in Level-2 M-file S-functions. These steps use the S-function `/msfcn_unit_delay.m`.

1 In the `PostPropagationSetup` method, initialize the number of DWork vectors and the attributes of each vector. For example, the following `PostPropagationSetup` callback method configures one DWork vector used to store a discrete state.

```
function PostPropagationSetup(block)

  %% Setup Dwork
  block.NumDworks                = 1;
  block.Dwork(1).Name            = 'x0';
  block.Dwork(1).Dimensions      = 1;
  block.Dwork(1).DatatypeID      = 0;
```

```
    block.Dwork(1).Complexity      = 'Real';
    block.Dwork(1).UsedAsDiscState = true;
```

The reference pages for `Simulink.BlockCompDworkData` and the parent class `Simulink.BlockData` list the properties you can set for Level-2 M-file S-function DWork vectors.

2 Initialize the DWork vector values in either the `Start` or `InitializeCondition` methods. Use the `Start` method for values that are initialized only at the beginning of the simulation. Use the `InitializeCondition` method for values that need to be reinitialized whenever a disabled subsystem containing the S-function is reenabled.

For example, the following `InitializeCondition` method initializes the value of the DWork vector configured in the previous step to the value of the first S-function dialog parameter.

```
function InitializeConditions(block)

  %% Initialize Dwork
  block.Dwork(1).Data = block.DialogPrm(1).Data;
```

3 In the `Outputs`, `Update`, etc. methods, use or update the DWork vector values, as needed. For example, the following `Outputs` method sets the S-function output equal to the value stored in the DWork vector. The `Update` method then changes the DWork vector value to the current value of the first S-function input port.

```
%% Outputs callback method
function Outputs(block)

  block.OutputPort(1).Data = block.Dwork(1).Data;

%% Update callback method
function Update(block)

  block.Dwork(1).Data = block.InputPort(1).Data;
```

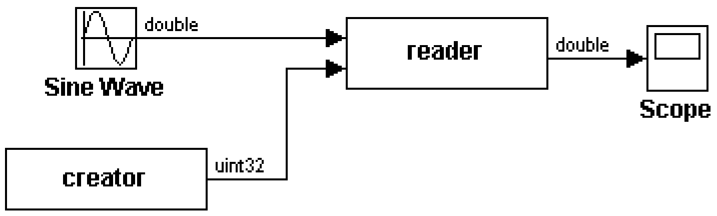
Note Level-2 M-file S-functions do not support MATLAB sparse matrices. Therefore, you cannot assign a sparse matrix to the value of a DWork vector. For example, the following line of code produces an error

```
block.Dwork(1).Data = speye(10);
```

where the `speye` command produces a sparse identity matrix.

Using DWork Vectors to Pass Data Between S-Functions

You can use DWork vectors to facilitate passing data from one S-functions to another. You do this by passing pointers, as unsigned integers, from the output port of one S-function to the input port of another S-function. For example, the following model contains two S-Function blocks, one calling the S-function `creator.c` and the other calling the S-function `reader.c`.



The S-function `creator.c` has one S-function dialog parameter that represents the gain value used by the S-function `reader.c`. The S-function `reader.c` outputs its input value multiplied by the gain value entered as the `creator.c` S-function dialog parameter.

For these two S-functions to communicate, the `mdlStart` method in `creator.c` allocates a shared piece of memory to store the S-function dialog parameter value and places the pointer to that memory in a DWork vector.

```
static void mdlStart(SimStruct *S)
{
    /* Get the value of the S-function dialog parameter */
    real_T GainVal = mxGetScalar(ssGetSFcnParam(S,0));
    /* Get the DWork vector, which is a pointer to a pointer */
    real_T **x = ssGetDWork(S,0);

    /* Allocate memory for the dialog parameter data. The S-function
     * owns this memory location. Simulink does not copy the data.
     * Place the pointer to this memory in the DWork vector. */
    real_T *gp = malloc(sizeof(GainVal));
    if (gp==NULL){
        ssSetErrorStatus(S,"Memory allocation error");
        return;
    }
    /* Assign a value into the new memory location */
    gp[0] = GainVal;

    /* Initialize the DWork vector */
    x[0] = gp;
}
```

The `mdlOutputs` method in `creator.c` outputs the pointer created in the `mdlStart` method as an unsigned integer (`uint32_T`).

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    uint32_T *y = ssGetOutputPortRealSignal(S,0);
    real_T **x = ssGetDWork(S, 0);
    real_T *gainAddr = x[0];

    /* Pass pointer as output */
    *y = (uint32_T)gainAddr;
}
```

The second input argument to `reader.c` accepts the `uint32_T` pointer value. The `mdlOutputs` method then accesses the value referenced by the pointer to use as the gain.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    /* Get the input signals. The second input is a pointer to
     * the pointer to the memory that holds the S-function dialog
     * parameter value */
```

```
        InputRealPtrsType uPtrs = ssGetInputPortSignal(S,0);
        uint32_T **pPtrs = ssGetInputPortSignal(S,1);

        /* Get information about the output signal */
        real_T       *y = ssGetOutputPortSignal(S,0);
        int_T     width = ssGetOutputPortWidth(S,0);
        int i;

        /* Retrieve the creator.c S-function dialog parameter value */
        uint32_T gainAddr = *pPtrs[0];
        real_T Gain = *(double*)gainAddr;

        /* Calculate outputs */
        for(i =0; i<width; i++)
        *y++ = (Gain)*(*uPtrs[i]);
    }
```

The `mdlTerminate` method in `creator.c` must free the memory created for the shared data.

```
    static void mdlTerminate(SimStruct *S)
    {
        real_T **x = ssGetDWork(S, 0);
        real_T *gainAddr = x[0];

        /* Free the memory */
        if (gainAddr != NULL) {
            free(gainAddr);
        }

        /* Clear out the DWork vector */
        x[0] = NULL;
    }
```

The previous method can be used to share any type of data between S-functions, for example, character arrays or structures.

Using DWork Vectors With Legacy Code

You can use DWork vectors to communicate with legacy code. If you have existing code that allocates data structures in memory, store a pointer to those data structures in a DWork vector. Your S-function can then communicate with the legacy code via the pointer. Alternatively, for simplicity in setting up your S-function, you can use a pointer work vector to store the pointer. See "Elementary Work Vectors" on page 7-29 for a description of pointer work vectors.

You can also use DWork vectors to store the state of legacy code. The template file `matlabroot/simulink/src/sfuntmpl_gate_fortran.c` shows how to use DWork vectors to interact with legacy Fortran code. The Legacy Code Tool uses DWork vectors to maintain the states of legacy C or C++ code incorporated through the tool. See "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" on page 4-55 for more information on the Legacy Code Tool.

DWork Vector Examples

In this section...

General DWork Vector

The S-function `sfun_rtwdwork.c` shows how to configure a DWork vector for use with the Real-Time Workshop product. The Simulink model `sfcndemo_sfun_rtwdwork.mdl` uses this S-function to implement a simple accumulator.

The following portion of the `mdlInitializeSizes` method initializes the DWork vector and all code generation properties associated with it.

```
ssSetNumDWork(S, 1);
ssSetDWorkWidth(S, 0, 1);
ssSetDWorkDataType(S, 0, SS_DOUBLE);

/* Identifier; free any old setting and update */
id = ssGetDWorkRTWIdentifier(S, 0);
if (id != NULL) {
    free(id);
}
id = malloc(80);
mxGetString(ID_PARAM(S), id, 80);
ssSetDWorkRTWIdentifier(S, 0, id);

/* Type Qualifier; free any old setting and update */
tq = ssGetDWorkRTWTypeQualifier(S, 0);
if (tq != NULL) {
    free(tq);
}
```

```
tq = malloc(80);
mxGetString(TQ_PARAM(S), tq, 80);
ssSetDWorkRTWTypeQualifier(S, 0, tq);

/* Storage class */
sc = ((int_T) *((real_T*) mxGetPr(SC_PARAM(S)))) - 1;
ssSetDWorkRTWStorageClass(S, 0, sc);
```

The S-function initializes the DWork vector in `mdlInitializeConditions`.

```
#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions ============================
 * Abstract:
 *    Initialize both continuous states to zero
 */
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *x = (real_T*) ssGetDWork(S,0);

    /* Initialize the dwork to 0
    x[0] = 0.0;
}
```

The `mdlOutputs` method assigns the DWork vector value to the S-function output.

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

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

The `mdlUpdate` method increments the DWork value by the input.

```
#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 = (real_T*) ssGetDWork(S,0);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

    /*
     * Increment the state by the input
     * U is defined as U(element) (*uPtrs[element])
     */
    x[0] += U(0);
}
```

DWork Scratch Vector

The following example uses a scratch DWork vector to store a static variable value. The `mdlInitializeSizes` method configures the width and data type of the DWork vector. The `ssSetDWorkUsageType` macro then specifies the DWork vector is a scratch vector.

```
ssSetNumDWork(S, 1);

ssSetDWorkWidth(S, 0, 1);
ssSetDWorkDataType(S, 0, SS_DOUBLE);
ssSetDWorkUsageType(S,0, SS_DWORK_USED_AS_SCRATCH);
```

The remainder of the S-function uses the scratch DWork vector exactly as it would any other type of DWork vector. The `InitializeConditions` method sets the initial value and the `mdlOutputs` method uses the value stored in the DWork vector.

```
#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions ================================
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *x = (real_T*) ssGetDWork(S,0);
    /* Initialize the dwork to 0 */
    x[0] = 0.0;
}
/* Function: mdlOutputs =============================================
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T *y = ssGetOutputPortRealSignal(S,0);
    real_T *x1 = (real_T*) ssGetDWork(S,1);

    x[0] = 2000;
    y[0] = x[0] * 2;
}
```

The Real-Time Workshop software handles scratch DWork differently from other DWork vectors when generating code for inlined S-function. To inline the S-function, create the following Target Language Compiler (TLC) file to describe the `mdlOutputs` method.

```
%implements sfun_dscratch "C"

%% Function: Outputs ==========================================================
%%
/* dscratch Block: %<Name> */
%<LibBlockDWork(DWork[0], "", "", 0)> = 2000.0;
%<LibBlockOutputSignal(0,"","",0)> = %<LibBlockDWork(DWork[0],"","", 0)> * 2;
```

When the Real-Time Workshop software generates code for the model, it inlines the S-function and declares the second DWork vector as a local scratch vector. For example, the model outputs function contains the following lines:

```
* local scratch DWork variables */
real_T SFunction_DWORK1;
SFunction_DWORK1 = 2000.0;
```

If the S-function used a general DWork vector instead of a scratch DWork vector, generating code with the same TLC file would have resulted in the DWork vector being included in the data structure, as follows:

```
sfcndemo_dscratch_DWork.SFunction_DWORK1 = 2000.0;
```

DState Work Vector

This example rewrites the S-function example `dsfunc.c` to use a DState vector instead of an explicit discrete state vector. The `mdlInitializeSizes` macro initializes the number of discrete states as zero and, instead, initializes one DWork vector.

The `mdlInitializeSizes` method then configures the DWork vector as a DState vector using a call to `ssSetDWorkUsedAsDState`. This is equivalent to calling the `ssSetDWorkUsageType` macro with the value `SS_DWORK_USED_AS_DSTATE`. The `mdlInitializeSizes` method sets the width and data type of the DState vector and gives the state a name using `ssSetDWorkName`.

Note DWork vectors configured as DState vectors must be assigned a name for the Simulink engine to register the vector as discrete states. The function `Simulink.BlockDiagram.getInitialStates(`*`mdl`*`)` returns the assigned name in the `label` field for the initial states.

```
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch reported by the Simulink engine*/
    }

    ssSetNumContStates(S, 0);
    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);

    ssSetNumDWork(S, 1);
    ssSetDWorkUsedAsDState(S, 0, SS_DWORK_USED_AS_DSTATE);
    ssSetDWorkWidth(S, 0, 2);
    ssSetDWorkDataType(S, 0, SS_DOUBLE);
    ssSetDWorkName(S, 0, "SfunStates");

    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}
```

The `mdlInitializeCondition` method initializes the DState vector values using the pointer returned by `ssGetDWork`.

```
#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions ===============================
 * Abstract:
 *    Initialize both discrete states to one.
 */
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *x0 = (real_T*) ssGetDWork(S, 0);
    int_T  lp;

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

The `mdlOutputs` method then uses the values stored in the DState vector to compute the output of the discrete state-space equation.

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

Finally, the `mdlUpdate` method updates the DState vector with new values for the discrete states.

```
#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       = (real_T*) ssGetDWork(S, 0);
    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];
}
```

DWork Mode Vector

This example rewrites the S-function `sfun_zc.c` to use a DWork mode vector instead of an explicit mode work vector (see “Elementary Work Vectors” on page 7-29 for more information on mode work vectors). This S-function implements an absolute value block.

The `mdlInitializeSizes` method sets the number of DWork vectors and zero-crossing vectors (see "Zero Crossings" on page 8-51) to `DYNAMICALLY_SIZED`. The `DYNAMICALLY_SIZED` setting allows the Simulink engine to defer specifying the work vector sizes until it knows the dimensions of the input, allowing the S-function to support an input with an arbitrary width.

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

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

    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);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumDWork(S, 1);
    ssSetNumModes(S, 0);

    /* Initializes the zero-crossing and DWork vectors
    ssSetDWorkWidth(S,0,DYNAMICALLY_SIZED);
    ssSetNumNonsampledZCs(S, DYNAMICALLY_SIZED);

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

The Simulink engine initializes the number of zero-crossing vectors and DWork vectors to the number of elements in the signal coming into the first S-function input port. The engine then calls the `mdlSetWorkWidths` method, which uses `ssGetNumDWork` to determine how many DWork vectors were initialized and then sets the properties for each DWork vector.

```
#define MDL_SET_WORK_WIDTHS
static void mdlSetWorkWidths(SimStruct *S) {
    int_T numdw = ssGetNumDWork(S);
    int_T i;

    for (i = 0; i < numdw; i++) {
        ssSetDWorkUsageType(S, i, SS_DWORK_USED_AS_MODE);
        ssSetDWorkDataType(S, i, SS_BOOLEAN);
        ssSetDWorkComplexSignal(S, i, COMPLEX_NO);
    }
}
```

The `mdlOutputs` method uses the value stored in the DWork mode vector to determine if the output signal should be equal to the input signal or the absolute value of the input signal.

```
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);
    boolean_T         *mode = ssGetDWork(S,0);

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

    if (ssIsMajorTimeStep(S)) {
        for (i = 0; i < width; i++) {
            mode[i] = (boolean_T)(*uPtrs[i] >= 0.0);
        }
    }

    for (i = 0; i < width; i++) {
        y[i] = mode[i]? (*uPtrs[i]): -(*uPtrs[i]);
    }
}
```

Level-2 M-File S-Function DWork Vector

The example S-function `msfcn_varpulse.m` models a variable width pulse generator. The S-function uses two DWork vectors. The first DWork vector stores the pulse width value, which is modified at every major time step in the `Update` method. The second DWork vector stores the handle of the pulse generator block in the Simulink model. The value of this DWork vector does not change over the course of the simulation.

The `PostPropagationSetup` method, called `DoPostPropSetup` in this S-function, sets up the two DWork vectors.

```
function DoPostPropSetup(block)

% Initialize the Dwork vector
block.NumDworks = 2;

% Dwork(1) stores the value of the next pulse width
block.Dwork(1).Name            = 'x1';
block.Dwork(1).Dimensions      = 1;
block.Dwork(1).DatatypeID      = 0;      % double
block.Dwork(1).Complexity      = 'Real'; % real
block.Dwork(1).UsedAsDiscState = true;

% Dwork(2) stores the handle of the Pulse Geneator block
block.Dwork(2).Name            = 'BlockHandle';
block.Dwork(2).Dimensions      = 1;
block.Dwork(2).DatatypeID      = 0;      % double
block.Dwork(2).Complexity      = 'Real'; % real
block.Dwork(2).UsedAsDiscState = false;
```

The `Start` method initializes the DWork vector values.

```
function Start(block)

% Populate the Dwork vector
block.Dwork(1).Data = 0;
```

```
% Obtain the Pulse Generator block handle
pulseGen = find_system(gcs,'BlockType','DiscretePulseGenerator');
blockH = get_param(pulseGen{1},'Handle');
block.Dwork(2).Data = blockH;
```

The `Outputs` method uses the handle stored in the second DWork vector to update the pulse width of the Pulse Generator block.

```
function Outputs(block)

% Update the pulse width value
set_param(block.Dwork(2).Data, 'PulseWidth', num2str(block.InputPort(1).data));
```

The `Update` method then modifies the first DWork vector with the next value for the pulse width, specified by the input signal to the S-Function block.

```
function Update(block)

% Store the input value in the Dwork(1)
block.Dwork(1).Data = block.InputPort(1).Data;

%endfunction
```

Elementary Work Vectors

In this section...

Description of Elementary Work Vector

In addition to DWork vectors, the Simulink software provides a simplified set of work vectors. In some S-functions, these elementary work vectors can provide an easier solution than using DWork vectors:

- **IWork vectors** store integer data.
- **Mode vectors** model zero crossings or other features that require a single mode vector.
- **PWork vectors** store pointers to data structures, such as those that interface the S-function to legacy code, another software application, or a hardware application.
- **RWork vectors** store floating-point (real) data.

Relationship to DWork Vectors

The following table compares each type of work vector to a DWork vector.

Work Vector Type	Comparison to DWork Vector	How to create equivalent DWork vector

IWork	IWork vectors cannot be customized in the generated code. Also, you are allowed only one IWork vector.	`ssSetNumDWork(S,1);`
		`ssSetDWorkDataType(S, 0, SS_INT8);`

Work Vector Type	Comparison to DWork Vector	How to create equivalent DWork vector

Mode	Mode vectors require more memory then DWork vectors since the mode vector is always stored with an `integer` data type. Also, you are allowed only one Mode vector.	`ssSetNumDWork(S,1);`
		`ssSetDWorkUsageType(S, 0,`
		`   sSS_DWORK_USED_AS_MODE);`
		`ssSetDWorkDataType(S, 0, SS_INT8);`

PWork	Unlike DWork vectors, PWork vectors cannot be named in the generated code. Also, you are allowed only one PWork vector.	`ssSetNumDWork(S,1);`
		`ssSetDWorkDataType(S, 0, SS_POINTER);`
		
		The DWork vector then stores a pointer.

RWork	RWork vectors cannot be customized in the generated code. Also, you are allowed only one RWork vector.	`ssSetNumDWork(S,1);`
		`ssSetDWorkDataType(S, 0, SS_DOUBLE);`

Using Elementary Work Vectors

The process for using elementary work vectors is similar to that for DWork vectors (see "Using DWork Vectors in C MEX S-Functions" on page 7-7.) The elementary work vectors have fewer properties, so the initialization process is simpler. However, if you need to generate code for the S-function, the S-function becomes more involved than when using DWork vectors.

The following steps show how to set up and use elementary work vectors. See "Additional Work Vector Macros" on page 7-32 for a list of macros related to each step in the following process.

1 In `mdlInitializeSizes`, specify the size of the work vectors using the `ssSetNumXWork` macro, for example:

```
ssSetNumPWork(2);
```

This macro indicates how many elements the work vector contains, however, the Simulink engine does not allocate memory, at this time.

An S-function can defer specifying the length of the work vectors until all information about the S-function inputs is available by passing the value `DYNAMICALLY_SIZED` to the `ssSetNumXWork` macro. If an S-function defers specifying the length of the work vectors in `mdlInitializeSizes`, it must provide a `mdlSetWorkWidths` method to set up the work vectors.

Note If an S-function uses `mdlSetWorkWidths`, all work vectors used in the S-function must be set to `DYNAMICALLY_SIZED` in `mdlInitializeSizes`, even if the exact value is known before `mdlInitializeSizes` is called. The sizes to be used by the S-function are than specified in `mdlSetWorkWidths`.

For an example, see `sfun_dynsize.c`.

---

2 In `mdlStart`, assign values to the work vectors that are initialized only at the start of the simulation. Use the `ssGetXWork` macro to retrieve a pointer to each work vector and use the pointer to initialize the work vector values. Alternatively, use the `ssGetXWorkValues` to assign values to particular elements of the work vector.

   The Simulink engine calls the `mdlStart` method once at the beginning of the simulation. Before calling this method, the engine allocates memory for the work vectors. Do not use the `mdlStart` method for data that needs to be reinitialized over the course of the simulation, for example, data that needs to be reinitialized when an enabled subsystem containing the S-function is enabled.

3 In `mdlInitializeConditions`, initialize the values of any work vectors that might need to be reinitialized at certain points in the simulation. The engine executes `mdlInitializeConditions` at the beginning of the simulation and any time an enabled subsystem containing the S-function is reenabled.

4 In `mdlOutputs`, `mdlUpdate`, etc., use the `ssGetXWork` macro to retrieve a pointer to the work vector and use the pointer to access or update the work vector values.

5 Write an `mdlRTW` method to allow the Target Language Compiler (TLC) to access the work vector. This step is not necessary if the S-function uses DWork vectors. See `ssWriteRTWParamSettings` for information on

writing parameter data in an `mdlRTW` method. See "Writing Fully Inlined S-Functions with the mdlRTW Routine" in the *Real-Time Workshop User's Guide* for more information on generating code using an `mdlRTW` method.

Additional Work Vector Macros

Macro	Description
`ssSetNumRWork`	Specify the width of the real work vector.
`ssGetNumRWork`	Query the width of the real work vector.
`ssSetNumIWork`	Specify the width of the integer work vector.
`ssGetNumIWork`	Query the width of the integer work vector.
`ssSetNumPWork`	Specify the width of the pointer work vector.
`ssGetNumPWork`	Query the width of the pointer work vector.
`ssSetNumModes`	Specify the width of the mode work vector.
`ssGetNumModes`	Query the width of the mode work vector.
`ssGetIWork`	Get a pointer to the integer work vector.
`ssGetIWorkValue`	Get an element of the integer work vector.
`ssGetModeVector`	Get a pointer to the mode work vector.
`ssGetModeVectorValue`	Get an element of the mode work vector.
`ssGetPWork`	Get a pointer to the pointer work vector.
`ssGetPworkValue`	Get one element from the pointer work vector.
`ssGetRWork`	Get a pointer to the floating-point work vector.
`ssGetRWorkValue`	Get an element of the floating-point work vector.
`ssSetIWorkValue`	Set the value of one element of the integer work vector.
`ssSetModeVectorValue`	Set the value of one element of the mode work vector.

Macro	Description
`ssSetPWorkValue`	Set the value of one element of the pointer work vector.
`ssSetRWorkValue`	Set the value of one element of the floating-point work vector.

Elementary Work Vector Examples

The following sections provide examples of the four types of elementary work vectors.

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 */
```

The following 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 Although the Simulink engine handles deallocating the PWork vector, the `mdlTerminate` method must always free the memory stored in the PWork vector.

Real and Integer Work Vectors

The S-function `stvctf.c` uses RWork and IWork vectors to model a time-varying continuous transfer function. For a description of this S-function, see the example "Discontinuities in Continuous States" on page 8-129.

Mode Vector

The following example implements a switch block using a mode work vector. The `mdlInitializeSizes` method configures two input ports with direct feedthrough and one output port. The mode vector element indicates if the signal from the first or second input port is propagated to the output. The S-function uses one S-function parameter and a corresponding run-time parameter to store the mode value and allow the switch to be toggled during simulation.

```
static void mdlInitializeSizes(SimStruct *S)
{
    /* Initialize one S-function parameter to toggle the mode value */
    ssSetNumSFcnParams(S, 1);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }
```

```
        {
            int iParam = 0;
            int nParam = ssGetNumSFcnParams(S);

            for ( iParam = 0; iParam < nParam; iParam++ )
            {
                ssSetSFcnParamTunable( S, iParam, SS_PRM_TUNABLE );
            }
        }
        /* Initialize two input ports with direct feedthrough */
        if (!ssSetNumInputPorts(S, 2)) return;
        ssSetInputPortWidth(S, 0, 1);
        ssSetInputPortWidth(S, 1, 1);
        ssSetInputPortDataType(  S, 0, SS_DOUBLE);
        ssSetInputPortDataType(  S, 1, SS_DOUBLE);
        ssSetInputPortDirectFeedThrough(  S, 0, 1);
        ssSetInputPortDirectFeedThrough(  S, 1, 1);

        /* Initialize one output port */
        if (!ssSetNumOutputPorts(S, 1)) return;
        ssSetOutputPortWidth(S, 0, 1);
        ssSetOutputPortDataType(  S, 0, SS_DOUBLE);

        /* Initialize one element in the mode vector */
        ssSetNumSampleTimes(S, 1);
        ssSetNumModes(S,1);

        ssSetOptions(S,
                     SS_OPTION_WORKS_WITH_CODE_REUSE |
                     SS_OPTION_USE_TLC_WITH_ACCELERATOR |
                     SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME |
                     SS_OPTION_NONVOLATILE);
    }
```

The `mdlInitializeConditions` method initializes the mode vector value using the current value of the S-function dialog parameter.

```
#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions =============================
 * Abstract:
 *    Initialize the mode vector value.
 */
static void mdlInitializeConditions(SimStruct *S)
{
    int_T *mv = ssGetModeVector(S);
    real_T param = mxGetScalar(ssGetSFcnParam(S,0));
    mv[0] = (int_T)param;
}
```

The `mdlProcessParameters` and `mdlSetWorkWidths` methods initialize and update the run-time parameter. As the simulation runs, changes to the S-function dialog parameter are mapped to the run-time parameter.

```
/* Function: mdlSetWorkWidths ============================================
 * Abstract:
 *    Sets the number of runtime parameters.
 */
#define MDL_SET_WORK_WIDTHS
static void mdlSetWorkWidths(SimStruct *S) {
    ssSetNumRunTimeParams(S,1);
    ssRegDlgParamAsRunTimeParam(S,0,0,"P1",SS_INT16);
}

/* Function: mdlProcessParameters ===========================================
 * Abstract:
 *      Update run-time parameters.
 */
#define MDL_PROCESS_PARAMETERS
static void mdlProcessParameters(SimStruct *S)
{
    ssUpdateDlgParamAsRunTimeParam(S,0);
}
```

The `mdlOutputs` method updates the mode vector value with the new run-time parameter value at every major time step. It then uses the mode vector value to determine which input signal to pass through to the output.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    InputRealPtrsType u2Ptrs = ssGetInputPortRealSignalPtrs(S,1);
    real_T    *y = ssGetOutputPortSignal(S,0);
    int_T  *mode = ssGetModeVector(S);
    real_T param = mxGetScalar(ssGetSFcnParam(S,0));

    if (ssIsMajorTimeStep(S)) {
        mode[0] =  (int_T)param;
    }

    if (!mode[0]) {
        /* first input */
        y[0] = (*uPtrs[0]);
    }
    if (mode[0]) {
        /* second input */
        y[0] = (*u2Ptrs[0]);
    }
}
```

Implementing Block Features

Dialog Parameters

In this section...

About Dialog Parameters

You can pass parameters to an S-function at the start of and during the simulation, using the **S-function parameters** field of the Block Parameters 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 8-8).

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

Using C S-Function Dialog Parameters

The Simulink engine stores the values of the dialog box parameters in the S-function's `SimStruct` structure. Use the S-function callback methods and `SimStruct` macros to access and check the parameters and use them to compute the S-function's output. To use dialog parameters in your C S-function, 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 the Simulink engine how many parameters the S-function accepts. Specify `S` as the first argument and the number of dialog box parameters 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. For example, the `mdlInitializeSizes` function in `sfun_runtime1.c` begins with the following code.

```
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 reported by the Simulink engine*/
    }
#endif
```

3 Access the dialog box parameters 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.

For example, in `sfun_runtime1.c`, the following `#define` statements at the beginning of the S-function specify the order of three dialog box parameters and access their values on the block's dialog.

```
#define SIGNS_IDX 0
#define SIGNS_PARAM(S) ssGetSFcnParam(S,SIGNS_IDX) /* First parameter */

#define GAIN_IDX  1
#define GAIN_PARAM(S) ssGetSFcnParam(S,GAIN_IDX) /* Second parameter */

#define OUT_IDX   2
#define OUT_PARAM(S) ssGetSFcnParam(S,OUT_IDX) /* Third parameter */
```

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

As another 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 at the top of the S-function 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, enter four variable names or values in the **S-function parameters** field of the S-Function Block Parameters 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);
```

Using Level-2 M-File S-Function Dialog Parameters

The Simulink engine stores Level-2 M-file S-function dialog parameters in the block's run-time object. To use dialog parameters in a Level-2 M-file S-function, 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 `setup` method, set the run-time object's `NumDialogPrms` property to indicate to the engine how many parameters the S-function accepts, for example:

   ```
   block.NumDialogPrms = 2;
   ```

3. Access the dialog box parameters in the S-function using the run-time object's `DialogPrm` method. The dialog parameter's `Data` property stores its current value, for example:

   ```
   param1 = block.DialogPrm(1).Data;
   param2 = block.DialogPrm(2).Data;
   ```

When running a simulation, you must specify the parameters in the **Parameters** field of the Level-2 M-file S-Function Block Parameters dialog box in the same order that you defined them in step 1.

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.

Note Dialog box parameters are tunable by default. Nevertheless, it is good programming practice to set the tunability of every parameter, even those that are tunable. If you enable the simulation diagnostic **S-function upgrades needed**, the Simulink engine 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. The engine invokes the `mdlCheckParameters` method whenever you change the values of parameters during the simulation loop. This method should check the S-function's dialog box parameters to ensure that the changes are valid.

The optional `mdlProcessParameters` callback method allows an S-function to process changes to tunable parameters. The engine 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 8-8).

Using Tunable Parameters in a C S-Function

In a C S-function, use the macro `ssSetSFcnParamTunable` in `mdlInitializeSizes` to specify the tunability of each S-function dialog box parameter. The code below is taken from the `mdlInitializeSizes` function in the example `sfun_runtime1.c`. The code first sets the number of S-function dialog box parameters to three before invoking `mdlCheckParameters`. If the parameter check passes, the tunability of the three S-function dialog box parameters is specified.

```
ssSetNumSFcnParams(S, 3); /* Three dialog box parameters*/

#if defined(MATLAB_MEX_FILE)
   if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
       mdlCheckParameters(S);
       if (ssGetErrorStatus(S) != NULL) {
           return;
       }
   } else {
       return; /* Parameter mismatch reported by the Simulink engine*/
   }
#endif

ssSetSFcnParamTunable(S,GAIN_IDX,true);   /* Tunable */
ssSetSFcnParamTunable(S,SIGNS_IDX,false); /* Not tunable */
ssSetSFcnParamTunable(S,OUT_IDX,false);   /* Not tunable */
```

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

Using Tunable Parameters in a Level-2 M-File S-Function

In a Level-2 M-file S-function, set the run-time object's `DialogPrmsTunable` property in the `setup` method to specify the tunability of each S-function dialog box parameter. For example, the following line sets the first parameter of an S-function with three dialog parameters to tunable, and the second and third parameters to nontunable.

```
block.DialogPrmsTunable = {'Tunable','Nontunable','Nontunable'};
```

.

Tuning Parameters in External Mode

When you tune parameters during simulation, the Simulink engine 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. The engine also invokes these methods when running in external mode, but it passes the unprocessed changes 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 "Inlining S-Functions" in the Real-Time Workshop Target Language Compiler documentation.

Run-Time Parameters

In this section...

About Run-Time Parameters

You can create internal representations of external S-function dialog box parameters called *run-time parameters*. Every run-time parameter corresponds to one or more dialog box 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. The Simulink engine 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 mass of the object. In this case, the mass 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. See "Creating Run-Time Parameters from Multiple S-Function Parameters" on page 8-12 for more information.

- 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 dialog parameter and the input and dialog 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, the Real-Time Workshop product 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.

The following Simulink model contains four example S-functions that create run-time parameters:

matlabroot`/toolbox/simulink/simdemos/simfeatures/sfcndemo_runtime.mdl`

Creating Run-Time Parameters

In a Level-2 M-file S-function, you create run-time parameters associated with all the tunable dialog parameters. Use the run-time object's `AutoRegRuntimePrms` method in the `PostPropagationSetup` callback method to register the block's run-time parameters. For example:

```
block.AutoRegRuntimePrms;
```

In a C S-function, you can create run-time parameters in a number of ways. The following sections describe different methods for creating run-time parameters in a C S-function.

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. The Real-Time Workshop product uses these names as the names of the parameters during code generation. The S-function *matlabroot*/simulink/src/sfun_runtime1.c shows how to create run-time parameters all at once.

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

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

Creating Run-Time Parameters Individually

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

1. Specify the number of run-time parameters it intends to use, using ssSetNumRunTimeParams.

2. Use ssRegDlgParamAsRunTimeParam to register a run-time parameter that corresponds to a single dialog parameter, even if there is a data type transformation, or ssSetRunTimeParamInfo to set the attributes of a run-time parameter that corresponds to more than one dialog parameter.

The following example uses ssRegDlgParamAsRunTimeParam and is taken from the S-function sfun_runtime3.c. This example creates a run-time parameter directly from the dialog parameter and with the same data type as the first input port's signal.

```
static void mdlSetWorkWidths(SimStruct *S)
{
    /* Get data type of input to use for run-time parameter */
    DTypeId      dtId         = ssGetInputPortDataType(S, 0);
    /* Define name of run-time parameter */
    const char_T *rtParamName = "Gain";

    ssSetNumRunTimeParams(S, 1); /* One run-time parameter */
    if (ssGetErrorStatus(S) != NULL) return;
    ssRegDlgParamAsRunTimeParam(S, GAIN_IDX, 0, rtParamName, dtId);
}
#endif /* MDL_SET_WORK_WIDTHS */
```

The next example uses `ssSetRunTimeParamInfo` and is taken from the S-function `sfun_runtime2.c`.

```
static void mdlSetWorkWidths(SimStruct *S)
{
    ssParamRec p; /* Initialize an ssParamRec structure */
    int        dlgP = GAIN_IDX; /* Index of S-function parameter */

    /* Configure run-time parameter information */
    p.name             = "Gain";
    p.nDimensions      = 2;
    p.dimensions       = (int_T *) mxGetDimensions(GAIN_PARAM(S));
    p.dataTypeId       = SS_DOUBLE;
    p.complexSignal    = COMPLEX_NO;
    p.data             = (void *)mxGetPr(GAIN_PARAM(S));
    p.dataAttributes   = NULL;
    p.nDlgParamIndices = 1;
    p.dlgParamIndices  = &dlgP;
    p.transformed      = false;
    p.outputAsMatrix   = false;

    /* Set number of run-time parameters */
    if (!ssSetNumRunTimeParams(S, 1)) return;

    /* Set run-time parameter information */
    if (!ssSetRunTimeParamInfo(S, 0, &p)) return;
}
```

The S-function `sfun_runtime2.c` defines the parameters `GAIN_IDX` and `GAIN_PARAM` as follows, prior to using these parameters in `mdlSetWorkWidths`.

```
#define GAIN_IDX  1
#define GAIN_PARAM(S) ssGetSFcnParam(S,GAIN_IDX)
```

Creating Run-Time Parameters from Multiple S-Function Parameters

Use the ssSetRunTimeParamInfo function in mdlSetWorkWidths to create run-time parameters as a function of multiple S-function parameters. For example, consider an S-function with two S-function parameters, density and volume. The S-function inputs a force (F) and outputs an acceleration (a). The mdlOutputs method calculates the force using the equation F=M/a, where the mass (M) is the product of the density and volume.

The S-function sfun_runtime4.c implements this example using a single run-time parameter to store the mass. The S-function begins by defining the run-time parameter data type, as well as variables associated with volume and density.

```
#define RUN_TIME_DATA_TYPE SS_DOUBLE
#if RUN_TIME_DATA_TYPE == SS_DOUBLE
typedef real_T RunTimeDataType;
#endif

#define VOL_IDX  0
#define VOL_PARAM(S) ssGetSFcnParam(S,VOL_IDX)

#define DEN_IDX   1
#define DEN_PARAM(S) ssGetSFcnParam(S,DEN_IDX)
```

The mdlSetWorkWidths method then initializes the run-time parameter, as follows.

```
static void mdlSetWorkWidths(SimStruct *S)
{
    ssParamRec p; /* Initialize an ssParamRec structure */
    int               dlg[2]; /* Stores dialog indices */
    real_T vol      = *mxGetPr(VOL_PARAM(S));
    real_T den      = *mxGetPr(DEN_PARAM(S));
    RunTimeDataType   *mass;

    /* Initialize dimensions for the run-time parameter as a
```

```
 * local variable. The Simulink engine makes a copy of this
 * information to store in the run-time parameter. */
int_T  massDims[2] = {1,1};

/* Allocate memory for the run-time parameter data. The S-function
 * owns this memory location. The Simulink engine does not copy the data.*/
if ((mass=(RunTimeDataType*)malloc(1)) == NULL) {
    ssSetErrorStatus(S,"Memory allocation error");
    return;
}

/* Store the pointer to the memory location in the S-function
 * userdata. Since the S-function owns this data, it needs to
 * free the memory during mdlTerminate */
ssSetUserData(S, (void*)mass);

/* Call a local function to initialize the run-time
 * parameter data. The Simulink engine checks that the data is not
 * empty so an initial value must be stored. */
calcMass(mass, vol, den);

/* Specify mass as a function of two S-function dialog parameters */
dlg[0] = VOL_IDX;
dlg[1] = DEN_IDX;

/* Configure run-time parameter information. */
p.name             = "Mass";
p.nDimensions      = 2;
p.dimensions       = massDims;
p.dataTypeId       = RUN_TIME_DATA_TYPE;
p.complexSignal    = COMPLEX_NO;
p.data             = mass;
p.dataAttributes   = NULL;
p.nDlgParamIndices = 2;
p.dlgParamIndices  = &dlg
p.transformed      = RTPARAM_TRANSFORMED;
p.outputAsMatrix   = false;

/* Set number of run-time parameters  */
if (!ssSetNumRunTimeParams(S, 1)) return;
```

```
    /* Set run-time parameter information */
    if (!ssSetRunTimeParamInfo(S,0,&p)) return;

}
```

The local function `calcMass` updates the run-time parameter value in `mdlSetWorkWidths` and in `mdlProcessParameters`, when the values of density or volume are tuned.

```
/* Function: calcMass ==============================================
 * Abstract:
 *      Local function to calculate the mass as a function of volume
 *      and density.
 */
static void calcMass(RunTimeDataType *mass, real_T vol, real_T den)
{
  *mass = vol * den;
}
```

The `mdlOutputs` method uses the stored mass to calculate the force.

```
/* Function: mdlOutputs ==========================================
 * Abstract:
 *
 *   Output acceleration calculated as input force divided by mass.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T *y1              = ssGetOutputPortRealSignal(S,0);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    RunTimeDataType *mass   =
        (RunTimeDataType *)((ssGetRunTimeParamInfo(S,0))->data);

     /*
     * Output acceleration = force / mass
     */
     y1[0] = (*uPtrs[0]) / *mass;
}
```

Lastly, the `mdlTerminate` method frees the memory allocated for the run-time parameter in `mdlSetWorkWidths`.

```
/* Function: mdlTerminate ==========================================
 * Abstract:
 *      Free the user data.
 */
static void mdlTerminate(SimStruct *S)
{
    /* Free memory used to store the run-time parameter data*/
   RunTimeDataType *mass = ssGetUserData(S);
    if (mass != NULL) {
        free(mass);
    }
}
```

To run the example, open the Simulink model:

```
sfcndemo_runtime.mdl
```

Updating Run-Time Parameters

Whenever you change the values of an S-function's dialog parameters during simulation, the Simulink engine invokes the S-function's `mdlCheckParameters` method to validate the changes. If the changes are valid, the engine 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.

In a Level-2 M-file S-function, update the run-time parameters using the `AutoUpdateRuntimePrms` method in the `ProcessParameters` callback method. For example:

```
block.AutoUpdateRuntimePrms;
```

In a C S-function, update the run-time parameters using the method appropriate for how the run-time parameters were created, as described in the following sections.

Updating All Parameters at Once

In a C MEX S-function, if there is a one-to-one correspondence between the S-function's tunable dialog parameters and the run-time parameters, i.e., the run-time parameters were registered using `ssRegAllTunableParamsAsRunTimeParams`, 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. See `sfun_runtime1.c` for an example.

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. Choose the method used to update the run-time parameter based on how it was registered.

If you register a run-time parameter using `ssSetRunTimeParamInfo`, the `mdlProcessParameters` method uses `ssUpdateRunTimeParamData` to update the run-time parameter, as shown in `sfun_runtime2.c`. This function updates the data field in the parameter's attributes record, `ssParamRec`, with a new value. You cannot directly modify the `ssParamRec`, even though you can obtain a pointer to the `ssParamRec` using `ssGetRunTimeParamInfo`.

If you register a run-time parameter using `ssRegDlgParamAsRunTimeParam`, the `mdlProcessParameters` method uses `ssUpdateDlgParamAsRunTimeParam` to update the run-time parameter, as is shown in `sfun_runtime3.c`.

Updating Parameters as Functions of Multiple S-Function Parameters

If you register a run-time parameter as a function of multiple S-function parameters, the `mdlProcessParameters` method uses `ssUpdateRunTimeParamData` to update the run-time parameter.

The S-function `sfun_runtime4.c` provides an example. In this example, the `mdlProcessParameters` method calculates a new value for the run-time parameter and passes the value to the pointer of the run-time parameter's memory location, which was allocated during the call to `mdlSetWorkWidths`.

The `mdlProcessParameters` method then passes the updated run-time parameter's pointer to `ssUpdateRunTimeParamData`.

Tuning Run-Time Parameters

Tuning a dialog parameter tunes the corresponding run-time parameter during simulation and in code generated only if the dialog parameter meets the following conditions:

- The S-function marks the dialog parameter as tunable, using `ssSetSFcnParamTunable`.
- The dialog parameter is a MATLAB array of values with a data type supported by the Simulink product.

Note that you cannot tune a run-time parameter whose value is a cell array or structure.

Accessing Run-Time Parameters

You can easily access run-time parameters from the S-function code. In order to access run-time parameter data, choose one of the following methods based on the data type.

- If the data is of type `double`:

  ```
  real_T *dataPtr = (real_T *) ssGetRunTimeParamInfo(S, #)->data;
  ```

- If the parameter is complex, the real and imaginary parts of the data are interleaved. For example, if a user enters the following:

  ```
  K = [1+2i, 3+4i; 5+6i, 7+8i]
  ```

  the matrix that is generated is

  ```
  K =
      1+2i     3+4i
      5+6i     7+8i
  ```

  The memory for this matrix is laid out as

  ```
  [1, 2, 5, 6, 3, 4, 7, 8]
  ```

To access a complex run-time parameter from the S-function code:

```
for (i = 0; i<width; i++)
{
real_T realData = dataPtr[(2*i)];
real_T imagData = dataPtr[(2*i)+1];
}
```

Note Matrix elements are written out in column-major format. Real and imaginary values are interleaved.

Input and Output Ports

In this section...

Creating Input Ports for C S-Functions

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

- The dimensions of the input port (see "Initializing Input Port Dimensions" on page 8-20)

  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 8-21).

- Whether the input port allows scalar expansion of inputs (see "Scalar Expansion of Inputs" on page 8-26)

- 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 the Simulink engine 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 `ssSetInputPortComplexSignal` 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 numeric type as `COMPLEX_INHERITED`. In this case, you must provide implementations of the `mdlSetInputPortComplexSignal` and `mdlSetDefaultPortComplexSignals` methods to enable the numeric type to be set correctly during signal propagation.

You can configure additional input port properties using other S-function macros. See "Input and Output Ports" on page 10-7 in the "SimStruct Macros and Functions Listed by Usage" section for more information.

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 a segmentation violation occurs.

Initializing Input Port Dimensions

You can set input port dimensions using one of the following macros:

- If the input signal must be one-dimensional and the input port width is `w`, use

  ```
  ssSetInputPortWidth(S, inputPortIdx, w)
  ```

- If the input signal must be a matrix of dimension `m`-by-`n`, use

  ```
  ssSetInputPortMatrixDimensions(S, inputPortIdx, m, n)
  ```

- Otherwise, if the input signal can have either one or two dimensions, 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 its input signals have specific dimensions, you can set the dimensionality of the input ports to match the dimensionality of the signals connected to them.

To dynamically dimension an input port:

- 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,
     DYNAMICALLY_SIZED, DYNAMICALLY_SIZED)
  ```

- Provide an `mdlSetInputPortDimensionInfo` method that sets the dimensions of the input port to the size of the signal connected to it.

  The Simulink engine 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. See `sfun_dynsize.c` for an example that implements this macro.

  The engine 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, the signal propagation routine sets the dimension of the block's ports to 1-D scalar.

Example: Defining Multiple S-Function Input Ports

The following code in `mdlInitializeSizes` configures an S-function with two input ports. See "Input and Output Ports" on page 10-7 in the "SimStruct Macros and Functions Listed by Usage" section for more information on the macros used in this example.

```
if (!ssSetNumInputPorts(S, 2)) return;

for (i = 0; i < 2; i++) {
  /* Input has direct feedthrough */
  ssSetInputPortDirectFeedThrough(S, i, 1);

  /* Input supports frames:
     Requires a Signal Processing Blockset license*/
  ssSetInputPortFrameData(S, i, FRAME_YES);

  /* Input is a real signal */
  ssSetInputPortComplexSignal(S, i, COMPLEX_NO);

  /* Input is a dynamically sized 2-D matrix */
  ssSetInputPortMatrixDimensions(S ,i,
     DYNAMICALLY_SIZED, DYNAMICALLY_SIZED);

  /* Input inherits its sample time */
  ssSetInputPortSampleTime(S, i,INHERITED_SAMPLE_TIME);

  /* Input signal must be contiguous */
  ssSetInputPortRequiredContiguous(S, i, 1);

  /* The input port cannot share memory */
```

```
    ssSetInputPortOverWritable(S, i, 0);
}
```

During signal propagation, the Simulink engine calls this S-function's `mdlSetInputPortDimensionInfo` macro to initialize the input port dimensions. In this example, `mdlSetInputPortDimensionInfo` sets the input dimensions to the candidate dimensions passed to the macro by the engine.

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_DIMENSION_INFO
static void mdlSetInputPortDimensionInfo(SimStruct        *S,
                                         int_T            port,
                                         const DimsInfo_T *dimsInfo)
{
    if(!ssSetInputPortDimensionInfo(S, port, dimsInfo)) return;
}
#endif
```

For an example that configures an S-function with multiple input and output ports, open the Simulink model `sfcndemo_sfun_multiport.mdl` and inspect the S-function `sfun_multiport.c`.

Creating Input Ports for Level-2 M-File S-Functions

To create and configure input ports, the `setup` method should first specify the number of S-function input ports, using the run-time object `NumInputPorts` property. Next, if all input ports inherit their functional properties (data type, dimensions, complexity, and sampling mode) from their input signals, include the following line in the `setup` method:

```
block.SetPreCompInpPortInfoToDynamic;
```

Then, for each input port, the `setup` method can specify

- The dimensions of the input port, using `block.InputPort(`*n*`).Dimensions`.

  To individually specify that an input port's dimensions are dynamically sized, assign a value of `-1` to the dimensions. In this case, you can implement the `SetInputPortDimensions` method to set the dimensions during signal propagation.

- Whether the input port has direct feedthrough, using `block.InputPort(n).DirectFeedthrough`.

  A port has direct feedthrough if the input is used in the `Outputs` functions to calculate either the outputs or the next sample time hit. The direct feedthrough flag for each input port can be set to either `1=yes` or `0=no`. Setting the direct feedthrough flag to `0` tells the Simulink engine that `u` is not used to calculate the outputs or next sample time hit. Violating this leads to unpredictable results.

- The data type of the input port, using `block.InputPort(n).DatatypeID`. See the explanation for the "DatatypeID" property in the `Simulink.BlockData` data object reference page for a list of valid data type IDs.

  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 `-1`. In this case, you can implement the `SetInputPortDataType` method to set the data type during signal propagation.

- The numeric type of the input port, if the port accepts complex-valued signals, using `block.InputPort(n).Complexity`.

  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 numeric type as `'Inherited'`. In this case, you can implement the `SetInputPortComplexSignal` method to set the numeric type during signal propagation.

- The sampling mode of the input port, using `block.InputPort(n).SamplingMode`.

  If you want the sampling mode of the port to depend on the sampling mode of the port to which it is connected, specify the sampling mode as `'Inherited'`. In this case, you can implement the `SetInputPortComplexSignal` method to set the sampling mode during signal propagation. If your S-function has multiple output ports, you must implement the `SetInputPortComplexSignal` method if any of the ports has an inherited sampling mode.

For an example that configures a Level-2 M-file S-function with multiple input and output ports, open the model `sldemo_msfcn_lms.mdl` and inspect the S-function `adapt_lms.c`.

Creating Output Ports for C S-Functions

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

- Dimensions of the output port

  You can set output port dimensions using one of the following macros:

  - `ssSetOutputPortDimensionInfo`
  - `ssSetOutputPortMatrixDimensions`
  - `ssSetOutputPortVectorDimension`
  - `ssSetOutputPortWidth`

  If you want the port's dimensions to depend on block connectivity, set the dimensions to `DYNAMIC_DIMENSIONS` when using `ssSetOutputPortDimensionInfo` or to `DYNAMICALLY_SIZED` for all other macros. The S-function must then provide `mdlSetOutputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` 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 output port, if the port outputs complex-valued signals

  Use `ssSetOutputPortComplexSignal` 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 numeric type as `COMPLEX_INHERITED`. In this case, you must provide implementations of the `mdlSetOutputPortComplexSignal` and `mdlSetDefaultPortComplexSignals` methods to enable the numeric type to be set correctly during signal propagation.

See “Creating Input Ports for C S-Functions” on page 8-19 for an example showing how to initialize an S-function input port. You use the same procedure to initialize the S-function output ports, but with the corresponding output port macro.

Creating Output Ports for Level-2 M-File S-Functions

To create output ports for Level-2 M-file S-functions the `setup` method should first specify the number of S-function output ports, using the run-time object `NumOutputPorts` property. Next, if all output ports inherit their functional properties (data type, dimensions, complexity, and sampling mode), include the following line in the `setup` method:

```
block.SetPreCompOutPortInfoToDynamic;
```

Configure the output ports exactly as you configure input ports. See “Creating Input Ports for Level-2 M-File S-Functions” on page 8-23 for a list of properties you can specify for each output port, substituting `OutputPort` for `InputPort` in each call to the run-time object.

Scalar Expansion of Inputs

Scalar expansion of inputs refers conceptually to the process of expanding scalar input signals to the same dimensions as wide input signals connected to other S-function input ports. This is done by setting each element of the expanded signal to the value of the scalar input.

- A Level-2 M-file S-function uses the default scalar expansion rules if the input and output ports are specified as dynamically sized (see “Scalar Expansion of Inputs and Parameters” in *Using Simulink*).
- A C MEX S-function’s `mdlInitializeSizes` method enables scalar expansion of inputs by setting the `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION` option, using `ssSetOptions`.

With scalar expansion on, the S-function’s `mdlInitializeSizes` method should specify that the input and output ports are dynamically sized. The Simulink engine uses a default method to set the dimensions of the input and output ports. If the block has more than two inputs, the input signals can be scalar or wide signals, where the wide signals all have the same number of elements. In this case, the engine sets the dimensions of the output ports to the width of the wide input signals and expands any scalar inputs to this width. 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.

If scalar expansion is not on, the engine 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.

Note The engine ignores the scalar expansion option if the S-function specifies or controls the dimensions of its input and output ports either by initializing the dimensions in `mdlInitializeSizes`, using `mdlSetInputPortWidth` and `mdlSetOutputPortWidth`, or using `mdlSetInputPortDimensionInfo`, `mdlSetOutputPortDimensionInfo`, and `mdlSetDefaultPortDimensionInfo`.

The best way to understand how to use scalar expansion is to consider the example *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/-sfcndemo_sfun_multiport.mdl`. This model contains three S-function blocks, each with multiple input ports. The S-function `sfun_multiport.c` used in these blocks sets the `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION` option in its `mdlInitializeSizes` method, allowing scalar expansion of the inputs. The S-function specifies that its inputs and outputs are dynamically sized. Therefore, during signal propagation, the engine sets the width of the input ports to the width of the signal connected to the port, and the width of the output ports to the width of any wide input signal. The `mdlOutputs` method performs an element-by-element sum on the input signals, expanding any scalar inputs, as needed.

```
/* Calculate an element-by-element sum of the input signals.
   yWidth is the width of the output signal. */

for (el = 0; el < yWidth; el++) {

   int_T  port;
   real_T sum = 0.0;
   for (port = 0; port < nInputPorts; port++) {
      /* Get the input signal value */
      InputRealPtrsType uPtrs =
                 ssGetInputPortRealSignalPtrs(S,port);
```

```
            if (el < ssGetInputPortWidth(S,port)) {
               /* Input is a wide signal. Use specific element */
               sum = sum + ((real_T)signs[port] * (*uPtrs[el]));

            } else {
               /* Use the scalar value to expand the signal */
               sum = sum + ((real_T)signs[port] * (*uPtrs[0]));
            }
        }
    }
```

Masked Multiport S-Functions

If you are developing masked multiport S-function blocks whose number of ports varies based on some parameter, and 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(blockname,'MaskSelfModifiable','on')
```

at the MATLAB command prompt before saving the library, where *blockname* is the full path to the block. 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

In this section...

Using Custom Data Types in C S-Functions

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`.

The following code placed at the beginning of `mdlInitializeSizes` sets the size and zero representation of a custom data type named `myDataType`.

```
/* Define variables */
int_T   status;
DTypeId  id;

/* Define the structure of the user-defined data type */
typedef struct{
    int8_T   a;
    uint16_T b;
}myStruct;

myStruct tmp;

/* Register the user-defined data types */
id = ssRegisterDataType(S, "myDataType");
if(id == INVALID_DTYPE_ID) return;

/* Set the size of the user-defined data type */
```

```
status = ssSetDataTypeSize(S, id, sizeof(tmp));
if(status == 0) return;

/* Set the zero representation */
tmp.a = 0;
tmp.b = 1;
status = ssSetDataTypeZero(S, id, &tmp);
```

To register a custom data type from a `Simulink.AliastType`, `Simulink.NumericType`, or `Simulink.StructType` object, the S-function's `mdlInitializeSizes` routine must:

1. Define an integer pointer to hold the data type identifier for the custom data type.

2. Register the data type, using `ssRegisterTypeFromNamedObject`.

For example, the following code placed at the beginning of `mdlInitializeSizes` defines a custom data type from a `Simulink.AliasType` object named `u8` in the MATLAB workspace. The example then assigns the custom data type to the first output port.

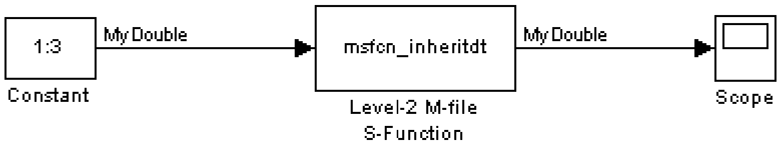
```
int id1;
ssRegisterTypeFromNamedObject(S, "u8", &id1);
ssSetOutputPortDataType(S, 0, id1);
```

In addition, you can use the identifier `id1` to assign this data type to S-function parameters, DWork vectors, and input ports.

Note You cannot use the Real-Time Workshop product to generate code for S-functions that contain macros to define custom data types. You must use an inlined S-function that accesses Target Language Compiler functions to generate code with custom data types. See "Inlining S-Functions" in the Real-Time Workshop Target Language Compiler documentation.)

Using Custom Data Types in Level-2 M-File S-Functions

Level-2 M-file S-functions do not support defining custom data types within the S-function. However, input and output ports can inherit their data types from a `Simulink.NumericType` or `Simulink.AliasType`. For example, the S-function in the following model inherits its input data type from the Constant block:



The Constant block's **Output data type** field contains the value `MyDouble`, which is a `Simulink.NumericType` defined in the MATLAB workspace with the following lines of code:

```
MyDouble = Simulink.NumericType;
MyDouble.IsAlias = 1;
```

The input and output ports of the Level-2 M-file S-function `msfcn_inheritdt.m` inherit their data types. When the Simulink engine performs data type propagation, it assigns the data type `MyDouble` to these ports.

You can define a fixed-point data type within a Level-2 M-file S-function, using one of the following three methods:

- `RegisterDataTypeFxpBinaryPoint` registers a fixed-point data type with binary point-only scaling
- `RegisterDataTypeFxpFSlopeFixExpBias` registers a fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias
- `RegisterDataTypeFxpSlopeBias` registers a data type with [Slope Bias] scaling

Note If the registered data type is not one of the Simulink built-in data types, you must have a Simulink® Fixed Point™ license.

Inspect the demo models and S-functions provided with the Simulink Fixed Point product for examples using the macros for defining fixed-point data types.

Sample Times

In this section...

About Sample Times

You can specify the sample-time behavior of your S-functions in `mdlInitializeSampleTimes`. Your S-function can inherit its rates from the blocks that drive it or define its own rates.

You can specify your S-function's 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 simulation 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 2 Hz and outputs at 4 Hz.

You should use port-based sample times if your application requires unequal sample rates for input and output execution or if you do not 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

Level-2 M-file S-functions specify block-based sample times in their `setup` method. Use the line

```
block.SampleTimes = [sampleTime offsetTime];
```

to specify the sample time. Use a value of `[-1 0]` to indicate an inherited sample time. See "How to Specify the Sample Time" in *Using Simulink* for a complete list of valid sample times.

C MEX S-functions specify block-based sample time information in

- `mdlInitializeSizes`
- `mdlInitializeSampleTimes`

The next two sections discuss how to specify block-based sample times for C MEX S-functions. A third section presents a simple example that shows how to specify sample times in `mdlInitializeSampleTimes`. For a detailed example, see `mixedm.c`.

Specifying the Number of Sample Times in mdlInitializeSizes

To configure your S-function for block-based sample times, use

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

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

Setting Sample Times and Specifying Function Calls in mdlInitializeSampleTimes

`mdlInitializeSampleTimes` specifies two pieces of execution information:

- Sample and offset times — In `mdlInitializeSampleTimes`, you must specify the sampling period and offset for each sample time using `ssSetSampleTime` and `ssSetOffsetTime`. If applicable, you can calculate the appropriate sampling period and offset prior to setting them, for example, by computing the best sample time for the block based on the S-function's dialog parameters obtained using `ssGetSFcnParam`.
- Function calls — In `mdlInitializeSampleTimes`, use `ssSetCallSystemOutput` to specify the output elements that are performing function calls. See`sfun_fcncall.c` for an example and "Function-Call Subsystems" on page 8-60 for an explanation of this S-function.

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`* and *`offsetTimePairIndex`* starts at 0.

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

```
[CONTINUOUS_SAMPLE_TIME,  0.0                      ]
[CONTINUOUS_SAMPLE_TIME,  FIXED_IN_MINOR_STEP_OFFSET]
[discrete_sample_period,  offset                    ]
```

```
[VARIABLE_SAMPLE_TIME  ,  0.0                       ]
```

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

```
[INHERITED_SAMPLE_TIME,  0.0                       ]
```

or

```
[INHERITED_SAMPLE_TIME,  FIXED_IN_MINOR_STEP_OFFSET]
```

Note If your S-function inherits its sample time, you should specify whether it is safe to use the S-function in a submodel, i.e., a model referenced by another model. See "Specifying Model Reference Sample Time Inheritance" on page 8-48 for more information.

The following guidelines might help in specifying sample times:

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

  ```
  [discrete_sample_period, offset]
  ```

  where

  ```
  discrete_sample_period > 0.0
  ```

  and

  ```
  0.0 <= offset < discrete_sample_period
  ```

- A discrete function that changes at a variable rate should register the variable-step discrete `[VARIABLE_SAMPLE_TIME, 0.0]` sample time. In C MEX S-functions, 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, use the following code fragment to check for a continuous sample hit:

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

To determine whether the third (discrete) task has a hit, use the following code fragment:

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

The Simulink engine always assigns an index of `0` to the continuous sample rate, if it exists, however you get incorrect results if you use `ssIsSampleHit(S,0,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

Port-based sample times cannot be used with S-functions that have neither input ports nor output ports. If an S-function uses port-based sample times and has no ports, the S-function produces errors when the Simulink model is updated or run. If the number of input or output ports on an S-function is variable, extra protection should be added into the S-function to ensure the total number of ports does not go to zero.

To use port-based sample times in a Level-2 M-file S-function:

- Specify the sample and offset times for each S-function port in the `setup` method. For example:

  ```
  block.InputPort(1).SampleTime = [-1 0];
  block.OutputPort(1).SampleTime = [-1 0];
  ```

  The `setup` method should not specify a sample time for the block when using port-based sample times.

- Provide `SetInputPortSampleTime` and `SetOutputPortSampleTime` methods, even if your S-function does not inherit its port-based sample times.

To use port-based sample times in your C MEX S-function, 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.

The call to `ssSetNumSampleTimes` can be placed before or after the port-based sample times are actually specified in `mdlInitializeSizes`. However, if `ssSetNumSampleTimes` does not configure the S-function to use port-based sample times, any sample times set on the ports will be ignored.

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 `0.1` and the offset time to `0`.

  ```
  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 8-39)
- Constant sample time, i.e., the port's input or output never changes (see "Specifying Constant Sample Time for a Port" on page 8-40)

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 8-42).

Specifying Inherited Sample Time for a Port

In a Level-2 M-file S-function, use a value of `[-1 0]` for the `SampleTime` property of each port to specify that the port inherits its sample time.

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

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

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

Once all rates have been determined, the engine calls `mdlInitializeSampleTimes`. Even though there is no need to initialize port-based sample times at this point, the engine 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. Use `ssGetInputPortSampleTime` and `ssGetOutputPortSampleTime` in `mdlInitializeSampleTimes` to obtain the values of the inherited sample times. For example, the following code in `mdlInitializeSampleTimes` checks if the S-function's first input inherited a continuous sample time.

```
if (!ssGetInputPortSampleTime(S,0)) {
    ssSetErrorStatus(S,"Cannot inherit a continuous sample time.")
};
```

Note If you specify that your S-function's ports inherit their sample time, you should also specify whether it is safe to use the S-function in a submodel, i.e., a model referenced by another model. See "Specifying Model Reference Sample Time Inheritance" on page 8-48 for more information.

Specifying Constant Sample Time for a Port

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

In a Level-2 M-file S-function, use the following line of code to specify a constant port-based sample time:

```
block.OutputPort(1).SampleTime = [inf 0];
```

The Simulink engine determines if a constant sample time is valid during sample-time propagation.

For C MEX S-functions, before specifying a 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 **Inline parameters** option on the **Optimization** pane of the **Configuration parameters** dialog box is on. If this option is not selected, you can change the values the S-function's parameters and hence its outputs during the simulation. In this case, the S-function can 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

- Use `ssSetOptions` to tell the Simulink engine 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 telling the engine that all of its ports support a constant sample time, including ports that inherit their sample times from other blocks. If any of the S-function's inherited sample time ports cannot have a constant sample time, your S-function's `mdlSetInputPortSampleTime` and `mdlSetOutputPortSampleTime` methods must check whether that port has inherited a constant sample time. If the port has inherited a constant sample time, your S-function should throw an error.

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

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

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

See `sfun_port_constant.c`, the source file for the `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

Level-2 M-file S-functions with port-based sample times cannot be placed in a triggered subsystem. You must modify your S-function to use block-based sample times if you need to include it in a triggered subsystem.

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

- Use `ssSetOptions` in the `mdlInitializeSizes` method to indicate the S-function can run in a triggered subsystem:

  ```
  ssSetOptions(S,
  SS_OPTION_ALLOW_PORT_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.

  Since the S-function's ports inherit their sample times, the Simulink engine invokes either `mdlSetInputPortSampleTime` or `mdlSetOutputPortSampleTime` during sample time propagation. The macro `ssSampleAndOffsetAreTriggered` can be used in these methods to determine if the S-function resides in a triggered subsystem. If the S-function does reside in a triggered subsystem, whichever method is called must set the sample time and offset of the port for which it is called to `INHERITED_SAMPLE_TIME (-1)`.

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 then 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, e.g.,

```
static void mdlSetInputPortSampleTime(SimStruct *S,
                                      int_T portIdx,
                                      real_T sampleTime
                                      real_T offsetTime)
{
    /* If the S-function resides in a triggered subsystem,
       the sample time and offset passed to this method
       are both equal to INHERITED_SAMPLE_TIME. Therefore,
       if triggered, the following lines set the sample time
       and offset of the input port to INHERITED_SAMPLE_TIME.*/

    ssSetInputPortSampleTime(S, portIdx, sampleTime);
    ssSetInputPortOffsetTime(S, portIdx, offsetTime);

    /* If triggered, set the output port to inherited, as well */

    if (ssSampleAndOffsetAreTriggered(sampleTime,offsetTime)) {
        ssSetOutputPortSampleTime(S, 0, INHERITED_SAMPLE_TIME);
        ssSetOutputPortOffsetTime(S, 0, INHERITED_SAMPLE_TIME);

        /* Note, if there are additional input and output ports
           on this S-function, they should be set to either
           inherited or constant at this point, as well. */
    }
}
```

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

- 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.mdl` demo model, for an example of how to create an S-function that can be used in a triggered subsystem.

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 block and input and output rates, using `ssSetNumSampleTimes`.

You then set the `SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED` option, 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. For an example S-function, see `mixedm.c`.

Note If you use the `SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED` option, your S-function cannot inherit sample times. Instead, you must specify the rate at which each input and output port runs.

Level-2 M-file S-functions do not support hybrid block-based and port-based sample times.

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. In a C MEX S-function, 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 in a C MEX S-function 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.

Use the following lines to encapsulate the code that defines the behavior for the continuous sample hit:

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

In a Level-2 M-file S-function, use the `IsSampleHit` method to determine whether the current simulation time is one at which a task handled by this block is active.

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 the Simulink engine 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);
```

Multirate S-Functions and Sample Time Hit Calculations

For fixed-step solvers, Simulink uses integer arithmetic, rather than floating-point arithmetic, to calculate the sample time hits. Consequently, task times are integer multiples of their corresponding sample time periods.

This calculation method becomes important if you consider performing Boolean logic based upon task times in multirate S-functions. For example, consider an S-function that has two sample times. The fact that (ssIsSampleHit(S, idx1) == true && ssIsSampleHit(S,idx2) == true, does not guarantee that ssGetTaskTime(S, idx1) == ssGetTaskTime(S, idx2).

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 (with a sample time index of 0) and an output port operating at a slower rate (with a sample time index of 1). Further, suppose that you want the output port to output the value currently on the input. The following example illustrates usage of this macro.

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

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

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

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

In a Level-2 M-file S-function, use the `IsSpecialSampleHit` method to determine whether the current simulation time is one at which multiple tasks implemented by this block are active.

Specifying Model Reference Sample Time Inheritance

If your C MEX S-function inherits its sample times from the blocks that drive it, it should specify whether submodels containing your S-function can inherit sample times from their parent model. If the S-function's output does not depend on its inherited sample time, use the `ssSetModelReferenceSampleTimeInheritanceRule` macro to set the S-function's sample time inheritance rule to `USE_DEFAULT_FOR_DISCRETE_INHERITANCE`. Otherwise, set the rule to `DISALLOW_SAMPLE_TIME_INHERITANCE` to disallow sample-time inheritance for submodels that include S-functions whose outputs depend on their inherited sample time and thereby avoid inadvertent simulation errors.

Note If your S-function does not set this flag, the Simulink engine assumes that it does not preclude a submodel containing it from inheriting a sample time. However, the engine optionally warns you that the submodel contains S-functions that do not specify a sample-time inheritance rule (see "Blocks Whose Outputs Depend on Inherited Sample Time" in *Using Simulink*).

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

- `ssGetSampleTime`

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

or TLC functions:

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

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

Sample-Time Inheritance Rule Example

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

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

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

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

Zero Crossings

S-functions model zero crossings using the mode work vector (or a DWork vector configured as a mode vector) and the continuous zero-crossing vector. Whether the S-function uses mode or DWork vectors, the concept and implementation is the same. For an example using DWork vectors to model zero crossings, see "DWork Mode Vector" on page 7-24 in the "Using Work Vectors" section. The remainder of this section uses mode vectors to model zero crossings.

Note Level-2 M-file S-functions do not support zero-crossing detection. The remainder of this section pertains only to C MEX S-functions.

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 values determine how the `mdlOutputs` routine operates when the solvers are homing in on zero crossings. The Simulink solvers track 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, by looking at the continuous zero crossings. Register the number of continuous zero crossings in `mdlInitializeSizes`, using `ssSetNumNonsampledZCs(S, num)`, then include an `mdlZeroCrossings` routine to calculate the continuous zero crossings. The S-function `sfun_zc_sat.c` contains a zero-crossing example. The remainder of this section describes the portions of this S-function that pertain to zero-crossing detection. For a full description of this example, see "Zero-Crossing Detection" on page 8-112.

First, `mdlInitializeSizes` specifies the sizes for the mode and continuous zero-crossing vectors using the following lines of code.

```
ssSetNumModes(S, DYNAMICALLY_SIZED);
ssSetNumNonsampledZCs(S, DYNAMICALLY_SIZED);
```

Since the number of modes and continuous zero crossings is dynamically sized, `mdlSetWorkWidths` must initialize the actual size of these vectors. In this example, shown below, there is one mode vector for each output element and two continuous zero crossings for each mode. In general, the number of continuous zero crossings needed for each mode depends on the number of events that need to be detected. In this case, each output (mode) needs to detect when it hits the upper or the lower bound, hence two continuous zero crossings per mode.

```
static void mdlSetWorkWidths(SimStruct *S)
{
    int nModes;
    int nNonsampledZCs;

    nModes         = numOutput;
    nNonsampledZCs = 2 * numOutput;

    ssSetNumModes(S,nModes);
    ssSetNumNonsampledZCs(S,nNonsampledZCs);
}
```

Next, `mdlOutputs` determines which mode the simulation is running in at the beginning of each major time step. The method stores this information in the mode vector so it is available when calculating outputs at both major and minor time steps.

```
/* Get the mode vector */
int_T *mode = ssGetModeVector(S);

    /* Specify three possible mode values.*/
    enum { UpperLimitEquation, NonLimitEquation, LowerLimitEquation };

    /* Update the mode vector at the beginning of a major time step */
    if ( ssIsMajorTimeStep(S) ) {
       for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {
            if ( *uPtrs[uIdx] > *upperLimit ) {
               /* Upper limit is reached. */
               mode[iOutput] = UpperLimitEquation;

            } else if ( *uPtrs[uIdx] < *lowerLimit ) {
               /* Lower limit is reached. */
               mode[iOutput] = LowerLimitEquation;

            } else {
               /* Output is not limited. */
```

```
                mode[iOutput] = NonLimitEquation;
            }

            /* Adjust indices to give scalar expansion. */
            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 */
```

Output calculations in `mdlOutputs` are done based on the values stored in the mode vector.

```
for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {
    if ( mode[iOutput] == UpperLimitEquation ) {
       /* Output upper limit. */
       *y++ = *upperLimit;

    } else if ( mode[iOutput] == LowerLimitEquation ) {
       /* Output lower limit. */
       *y++ = *lowerLimit;

    } else {
       /* Output is equal to input */
       *y++ = *uPtrs[uIdx];
    }
```

After outputs are calculated, the Simulink engine calls `mdlZeroCrossings` to determine if a zero crossing has occurred. A zero crossing is detected if any element of the continuous zero-crossing vector switches from negative to positive, or positive to negative. If this occurs, the simulation modifies the step size and recalculates the outputs to try to locate the exact zero crossing. For this example, the values for the continuous zero-crossing vectors are calculated as shown below.

```
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 );
    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 );

    /*Check if the input has crossed an upper or lower limit */
    for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {
        zcSignals[2*iOutput] = *uPtrs[uIdx] - *upperLimit;
        zcSignals[2*iOutput+1] = *uPtrs[uIdx] - *lowerLimit;

        /* Adjust indices to give scalar expansion if needed */
        uIdx       += uInc;
        upperLimit += upperLimitInc;
        lowerLimit += lowerLimitInc;
    }
}
```

S-Function Compliance with the `SimState`

In this section...

`SimState` Compliance Specification for Level-2 M-File S-Functions

In order for a Level-2 M-file S-function to work with the `SimState` feature, you must specify the `simStateCompliance` of the block using the method,

```
block.simStateCompliance = setting
```

where the permissible setting values are:

Setting	Result
`'UnknownSimState'`	This default setting instructs Simulink to use the `DefaultSimState` to save and restore the `SimState` and issues a warning.
`'DefaultSimState'`	This setting instructs Simulink to treat the S-function like a built-in block when saving and restoring the `SimState`.
`'HasNoSimState'`	This setting informs Simulink that the S-function does not have any simulation state. With this setting, no state information is saved for the block. This setting is primarily useful for "sink" blocks (i.e., blocks with no output ports) that use `PWorks` or `DWorks` to store handles to files or figure windows.
`'CustomSimState'`	This setting informs Simulink that the S-function has custom `GetSimState` and `SetSimState` methods.
`'DisallowSimState'`	This setting informs Simulink that the S-function does not allow saving or restoring its simulation state. Simulink reports an error if you save and restore the `SimState` of the model that contains this S-function.

For an S-function with custom methods (`'CustomSimState'`), you can use the following statements to respectively get and set the `SimState`:

```
function outSS = GetSimState(block)
function SetSimState(block, inSS)
```

For an example of how to implement these custom methods, see `msfcn_varpulse.m`.

`SimState` Compliance Specification for C-MEX S-Functions

As with the M-file S-function, your C-MEX S-function code must inform Simulink of the S-function's compliance with the `SimState` feature. You can accomplish this task by using the S-function API, `ssSetSimStateCompliance`.

In most cases, you must add only the following line:

```
ssSetSimStateCompliance(S, USE_DEFAULT_SIM_STATE).
```

The options are as follows:

Setting	Result
SIM_STATE_COMPLIANCE_UNKNOWN	This is the default setting for all S-functions. For S-functions that do not use PWorks, Simulink saves and restores the default simulation state (see next option) and issues a warning to inform the user of this assumption. On the other hand, Simulink reports an error during the save and restore if it encounters an S-function that uses PWorks.
USE_DEFAULT_SIM_STATE	This setting instructs Simulink to treat the S-function like a built-in block when saving and restoring the SimState.
HAS_NO_SIM_STATE	This setting informs Simulink that the S-function does not have any simulation state. With this setting, no state information is saved for this block. This setting is primarily useful for "sink" blocks (i.e., blocks with no output ports) that use PWorks or DWorks to store handles to files or figure windows.
DISALLOW_SIM_STATE	This setting informs Simulink that the S-function does not allow the saving or restoring of its simulation state. Simulink reports an error if you save and restore the SimState of the model that contains this S-function.
USE_CUSTOM_SIM_STATE	This setting informs Simulink that the S-function has mdlGetSimState and mdlSetSimState methods.

For S-functions that use PWork vectors or static variables to hold data that Simulink updates during simulation, the S-function must use the custom mdlGetSimState and mdlSetSimState methods. The following statements demonstrate the proper format.

```
mxArray* mdlGetSimState(SimStruct* S)
void mdlSetSimState(SimStruct* S, const mxArray* inSS)
```

For an example of how to implement these methods, see sfun_simstate.c.

Matrices in C S-Functions

In this section...
"MX Array Manipulation" on page 8-58
"Memory Allocation" on page 8-59

MX Array Manipulation

S-functions can manipulate `mxArrays` using the standard MATLAB API functions. (See "MX Library" in the MATLAB C and Fortran API Reference pages for a complete list of functions.) In general, if your S-function is declared exception free by passing the `SS_OPTION_EXCEPTION_FREE_CODE` option to `ssSetOptions` (see Exception Free Code in "Error Handling" on page 8-69), it should avoid MATLAB API functions that throw exceptions (i.e., long jump), such as `mxCreateDoubleMatrix`. Otherwise, the S-function can use any of the listed functions.

The Real-Time Workshop product supports a subset of the `mxArray` manipulation functions when generating noninlined code for an S-function. For a list of supported functions, see "Writing Noninlined S-Functions" in the *Real-Time Workshop User's Guide*.

Calls to the macro `ssGetSFcnParam` return a pointer to an `mxArray`, which can be used with the `mxArray` manipulation functions. If your S-function contains S-function parameters, use the `mxArray` manipulation functions in the `mdlCheckParameters` method to check the S-function parameter values. See the S-function `sfun_runtime3.c` for an example

In this S-function, the following lines check that the first S-function parameter is a character array with a length greater than or equal to two.

```
if (!mxIsChar(ssGetSFcnParam(S, 0)) ||
     (nu=mxGetNumberOfElements(ssGetSFcnParam(S, 0))) < 2) {
     ssSetErrorStatus(S,"1st parameter to S-function must be a "
       "string of at least 2 '+' and '-' characters");
     return;
}
```

Memory Allocation

When you create an S-function, you might 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 the MATLAB environment and not the Simulink environment. The correct approach for allocating memory is to use the `stdlib.h` library routines `calloc` and `free`. In `mdlStart`, allocate and initialize the memory

```
UD *ptr = (UD *)calloc(1,sizeof(UD));
```

where `UD`, in this example, is a data structure defined at the beginning of the S-function. Then, place the pointer to it either in the pointer work vector

```
ssSetPWorkValue(S, 0, ptr);
```

or attach it as user data.

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

In `mdlTerminate`, free the allocated memory. For example, if the pointer was stored in the user data

```
UD *prt = ssGetUserData(S);
free(prt);
```

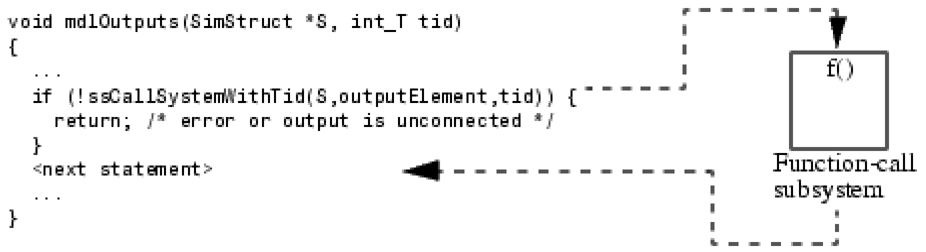
Function-Call Subsystems

You can create a triggered subsystem whose execution is determined by logic internal to a C MEX S-function instead of by the value of a signal. A subsystem so configured is called a *function-call subsystem*. You cannot trigger a function-call subsystem from a Level- M-file S-function. To implement a function-call subsystem:

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

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

Function-call subsystems are not executed directly by the Simulink engine; 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:

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

  ```
  ssSetCallSystemOutput(S,0);  /* call on first element */
  ssSetCallSystemOutput(S,1);  /* call on second element */
  ```

- Specify in `mdlInitializeSampleTimes` whether you want the S-function to be able to enable or disable the function-call subsystem. Only S-functions that explicitly enable and disable the function-call subsystem can reset the states and outputs of the subsystem, as determined by the function-call subsystem's Trigger and Outport blocks. For example, the code

  ```
  ssSetExplicitFCSSCtrl(S, 1);
  ```

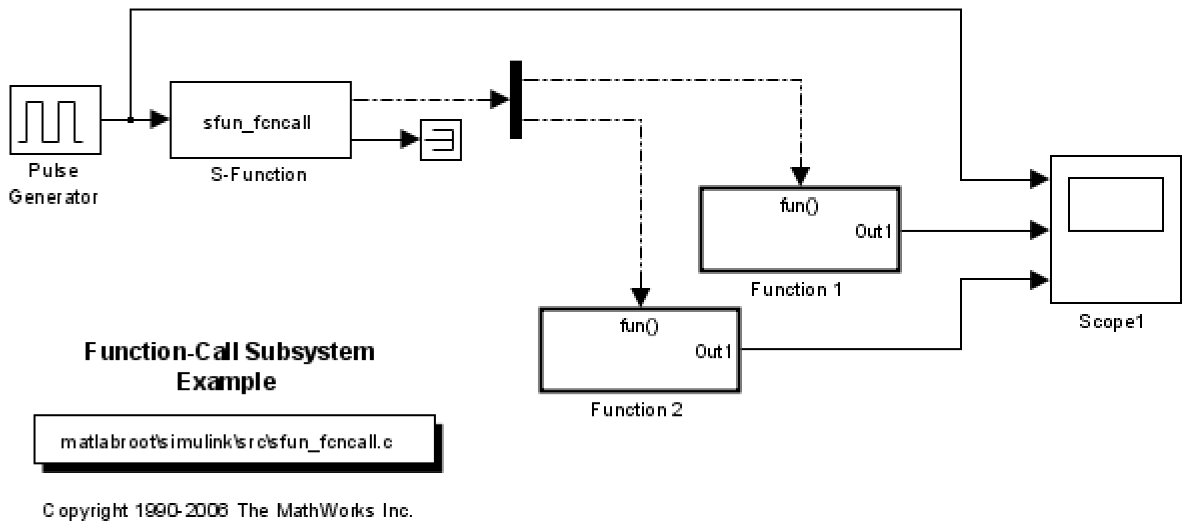
  in `mdlInitializeSampleTimes` specifies that the S-function can enable and disable the function-call subsystem. In this case, the S-function must invoke `ssEnableSystemWithTid` before executing the subsystem using `ssCallSystemWithTid`.

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

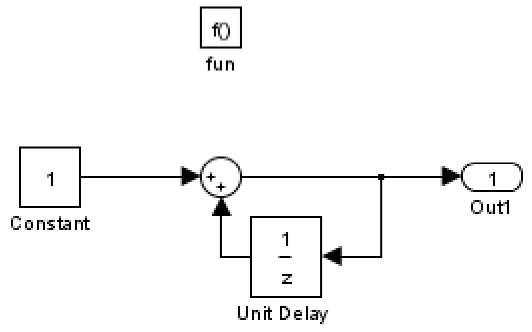
```
static void mdlOutputs(...)
{
    if (((int)*uPtrs[0]) % 2 == 1) {
      if (!ssCallSystemWithTid(S,0,tid)) {
        /* Error occurred, which will be reported by */
    /*the Simulink engine*/
        return;
      }
    } else {
      if (!ssCallSystemWithTid(S,1,tid)) {
        /* Error occurred,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

Note Do not use `ssSetOutputPortDataType` or `ssGetOutputPortDataType` on an S-function output that emits function-call signals. The Simulink engine explicitly controls the data type of these output signals.

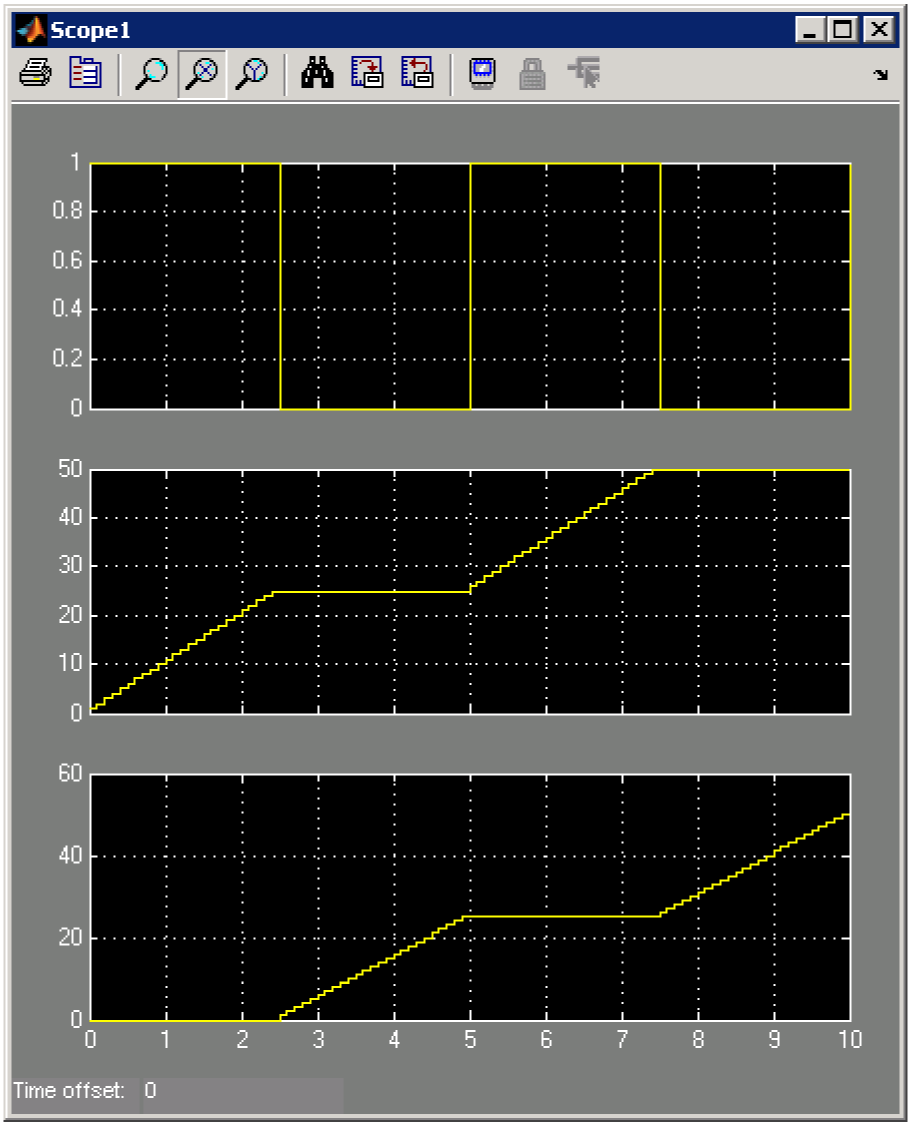
See `sfun_fcncall.c` for an example that executes a function-call subsystem on the first and second elements of the S-function's first output. The following Simulink model uses this S-function.


Pulse Generator
sfun_fcncall
S-Function
fun()
Out1
Function 1
fun()
Out1
Function 2
Scope1
Function-Call Subsystem Example
matlabroot\simulink\src\sfun_fcncall.c
Copyright 1990-2006 The MathWorks Inc.

Each of the function-call subsystems is a simple feedback loop containing a Unit Delay block, as shown below.



When the Pulse Generator emits its upper value, the function-call subsystem connected to the first element of the S-function's first output port is triggered. Similarly, when the Pulse Generator emits its lower value, the function-call subsystem connected to the second element is triggered. The simulation output is shown on the following Scope.



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, see the Stateflow documentation.

Sim Viewing Devices in External Mode

A sim viewing device encapsulates processing and viewing of signals received from the target system in external mode. During simulation in external mode, the target system uploads the appropriate input values to the sim viewing device in the Simulink model. The sim viewing device then conditions the input signals as needed and renders the signals on the screen. A sim viewing device runs only on the host, generating no code in the target system and, therefore, allowing extra processing of displayed signals without burdening the generated code.

You can use your S-function as a sim viewing device in external mode if it satisfies the following conditions.

- The S-function has no output ports.
- The S-function contains no states.
- The generated code does not require the conditioned signals produced by the S-function.

To specify a C MEX S-function is a sim viewing device, set the `SS_OPTION_SIM_VIEWING_DEVICE` option in the `mdlInitializeSizes` function. For example

```
ssSetOptions(S, SS_OPTION_SIM_VIEWING_DEVICE);
```

To specify a Level-2 M-file S-function is a sim viewing device, call the run-time object's `SetSimViewingDevice` method in the S-function's `setup` callback method.

External mode compatible S-functions are selected, and the trigger is armed, by using the External Signal & Triggering dialog box. For more information see "Communicating With Code Executing on a Target System Using Simulink External Mode" in the Real-Time Workshop documentation.

Frame-Based Signals

In this section...

About Frame-Based Signals

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

Note Simulating a model containing an S-function that accepts or produces frames requires a Signal Processing Blockset product license.

Using Frame-Based Signals in C S-Functions

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

- The S-function's `mdlInitializeSizes` callback method must set the port frame status to `FRAME_YES`, `FRAME_NO`, or `FRAME_INHERITED` for each of the S-function's I/O ports, using the `ssSetInputPortFrameData` and `ssSetOutputPortFrameData` functions. The frame status for a port must be set after the call to `ssSetNumInputPorts` and `ssSetNumOutputPorts`. For example, the following code in `mdlInitializeSizes` specifies that the first input port accepts a frame-based signal while the first output port emits a sample-based signal:

```
ssSetNumInputPorts(S, 1);
ssSetInputPortFrameData(S, 0, FRAME_YES);
ssSetNumOutputPorts(S,1);
ssSetOutputPortFrameData(S, 0, FRAME_NO);
```

- The S-function should specify the dimensions of the signals that its frame-based ports accept or produce in its `mdlInitializeSizes` or `mdlSetInputPortDimensionInfo` and `mdlSetOutputPortDimensionInfo` callback methods. Note that frame-based signals must be dimensioned as 2-D arrays. For example, the following code in `mdlInitializeSizes` specifies that the first frame-based input port is dynamically sized. This S-function must then also have an `mdlSetInputPortDimensionInfo` callback that sets the specific dimensions of this input port.

  ```
  ssSetNumInputPorts(S, 1);
  ssSetInputPortFrameData(S, 0, FRAME_YES);
  ssSetInputPortMatrixDimensions(S, 0, DYNAMICALLY_SIZED, DYNAMICALLY_SIZED);
  ```

- If the frame status of any of the S-function's input ports is inherited, the S-function should define a `mdlSetInputPortFrameData` callback method. The Simulink engine passes the frame status that it assigns to the port, based on frame signal propagation rules, as an argument to this callback method. The callback method should in turn use the `ssSetInputPortFrameData` function to set the port to the assigned status if it is acceptable or signal an error using `ssSetErrorStatus` if it is not. If the frame status of other ports of the S-function depend on the status inherited by one of its input ports, the callback method can also use `ssSetInputPortFrameData` to set the frame status of the other ports based on the status that the input port inherits. A template for the `mdlSetInputPortFrameData` callback is shown below.

  ```
  #if defined(MATLAB_MEX_FILE)
  #define MDL_SET_INPUT_PORT_FRAME_DATA
  static void mdlSetInputPortFrameData(SimStruct *S,
                                       int_T       portIndex,
                                       Frame_T   frameData)
  {
     if(!frameData==FRAME_YES) {
           ssSetErrorStatus(S, "Incorrect frame status");
           return;
     }
     ssSetInputPortFrameData(S, portIndex, frameData); /* Sets frame status */

  } /* end mdlSetInputPortFrameData */
  #endif
  ```

- The S-function's `mdlOutputs` method should include code to process the signals. The macro `ssGetInputPortDimensions` can be used in `mdlOutputs` to determine the dimensions of dynamically sized frame-based inputs, as follows:

  ```
  int *dims   = ssGetInputPortDimensions(S, 0);
  int frameSize = dims[0];
  int numChannels  = dims[1];
  ```

See the frame-based A/D converter S-function example (`sfun_frmad.c`) for an example of how to create a frame-based S-function. This S-function is one of several S-functions that manipulate frame-based signals found in the Simulink model `sfcndemo_frame.mdl`.

Using Frame-Based Signals in Level-2 M-File S-Functions

In a Level-2 M-file S-function, set the `SamplingMode` property of the port to indicate if the block accepts frame-based signals, for example:

```
block.InputPort(1).SamplingMode = 'Inherited';
```

If any of the ports inherited their sampling mode, define a `SetInputPortSamplingMode` callback method to specify the sampling mode.

Error Handling

In this section...

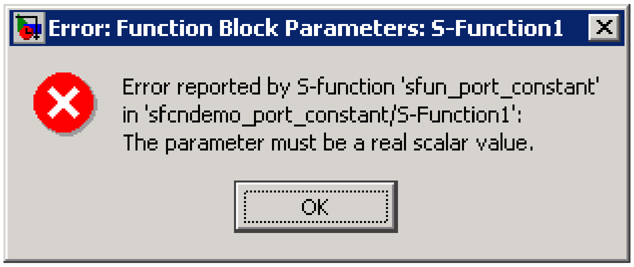
About Handling Errors

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

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

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

In most cases, the Simulink engine displays error messages in the Simulation Diagnostics Viewer. If the error is encountered in `mdlCheckParameters` as the S-function parameters are being entered into the block dialog, the engine opens the error dialog shown below. In either case, the engine displays the error message along with the name of the S-function and the associated S-function block that invoked the error.


Error: Function Block Parameters: S-Function1

Error reported by S-function 'sfun_port_constant'
in 'sfcndemo_port_constant/S-Function1':
The parameter must be a real scalar value.

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

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

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

Exception Free Code

You can avoid simulation overhead by ensuring that your C MEX 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 the Simulink engine 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 API routines 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 the engine calls during the simulation loop (see “How the Simulink Engine Interacts with C S-Functions” on page 4-77). 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

If one of your C MEX S-function’s callback methods invokes `ssSetErrorStatus` during a simulation, the Simulink engine posts the error and terminates the simulation as soon as the callback method returns. If your S-function’s `SS_OPTION_CALL_TERMINATE_ON_EXIT` option is enabled (see `ssSetOptions`), The engine invokes your S-function’s `mdlTerminate` method as part of the termination process. Otherwise, the engine invokes your S-function’s `mdlTerminate` method only if at least one block `mdlStart` method has executed without error during the simulation.

Checking Array Bounds

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

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

To enable array bounds checking, select `warning` or `error` from the **Array bounds exceeded** options list in the **Debugging** group on the **Diagnostics - Data Validity** pane of the **Configuration Parameters** dialog box or enter the following command at the MATLAB command prompt.

```
set_param(modelName, 'ArrayBoundsChecking', ValueStr)
```

where *`modelName`* is the name of the Simulink model and *`ValueStr`* is either `'none'`, `'warning'`, or `'error'`.

C MEX S-Function Examples

In this section...

About S-Function Examples

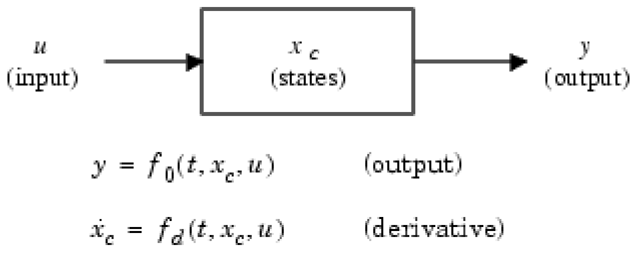
All examples are based on the C MEX-file S-function templates `sfuntmpl_basic.c` and `sfuntmpl_doc.c`. Open `sfuntmpl_doc.c`. for a detailed discussion of the S-function template.

Continuous States

The *`matlabroot`*`/simulink/src/csfunc.c` example shows how to model a continuous system with states using a C MEX S-function. The following Simulink model uses this S-function.

`matlabroot``/toolbox/simulink/simdemos/simfeatures/sfcndemo_csfunc.mdl`

In continuous state integration, the Simulink solvers integrate a set of continuous states using the following equations.



u (input)
x_c (states)
y (output)
$y = f_0(t, x_c, u)$ (output)
$\dot{x}_c = f_d(t, x_c, u)$ (derivative)

S-functions that contain continuous states implement a state-space equation. The `mdlOutputs` method contains the output portion and `mdlDerivatives` method contains the derivative portion of the state-space equation. To visualize how the integration works, see the flowchart in "How the Simulink Engine Interacts with C S-Functions" on page 4-77. The output equation corresponds to the `mdlOutputs` in the major time step. Next, the example enters the integration section of the flowchart. Here the Simulink engine 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.

The `csfunc.c` example 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 is not 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

The S-function begins with `#define` statements for the S-function's name and level, and a `#include` statement for the `simstruc.h` header. After these statements, the S-function can include or define any other necessary headers, data, etc. The `csfunc.c` example defines the variable `U` as a pointer to the first input port's signal and initializes static variables for the state-space matrices.

```
/*  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 1990-2007 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     }
                      };
```

The required S-function method `mdlInitializeSizes` then sets up the following S-function characteristics.

- `ssSetNumSFcnParams` sets the number of expected S-function dialog parameters to zero.

- ssGetSFcnParamsCount determines how many parameters the user actually entered into the S-function dialog. If the number of user-specified parameters does not match the number returned by ssGetNumSFcnParams, the S-function errors out.
- If the S-function parameter count passes, mdlInitializeSizes sets the number of continuous and discrete states using ssSetNumContStates and ssSetNumDiscStates, respectively. This example has two continuous states and zero discrete states.
- Next, the method configures the S-function to have a single input and output port, each with a width of two to match the dimensions of the state-space matrices. The method passes a value of 1 to ssSetInputPortDirectFeedThrough to indicate the input port has direct feedthrough.
- ssSetNumSampleTimes initializes one sample time, which the mdlInitializeSampleTimes function configures later.
- The S-function indicates that no work vectors are used by passing a value of 0 to ssSetNumRWork, ssSetNumIWork, etc. These lines could be omitted because zero is the default value for all of these macros. However, for clarity, the S-function explicitly sets the number of work vectors.
- Lastly, ssSetOptions sets any applicable options. In this case, the only option is SS_OPTION_EXCEPTION_FREE_CODE, which stipulates that the code is exception free.

The mdlInitializeSizes function for this example is shown below.

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

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    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 reported by the Simulink engine*/
    }

    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);
}
```

The required S-function method `mdlInitializeSampleTimes` specifies the S-function's sample rates. The value `CONTINOUS_SAMPLE_TIME` passed to the `ssSetSampleTime` macro specifies that the S-function's first sample rate is continuous. `ssSetOffsetTime` then specifies an offset time of zero for this sample rate. The call to `ssSetModelReferenceSampleTimeDefaultInheritance` tells the solver to use the default rule to determine if submodels containing this S-function can inherit their sample times from the parent model.

```
/* 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);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
}
```

The optional S-function method `mdlInitializeConditions` initializes the continuous state vector. The `#define` statement before this method is required for the Simulink engine to call this function. In the example below, `ssGetContStates` obtains a pointer to the continuous state vector. The `for` loop then initializes each state to zero.

```
#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;
    }
}
```

The required `mdlOutputs` function computes the output signal of this S-function. The beginning of the function obtains pointers to the first output port, continuous states, and first input port. The S-function uses the data in these arrays to solve the output equation `y=Cx+Du`.

```
/* 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);
}
```

The `mdlDerivatives` function calculates the continuous state derivatives. Because this function is an optional method, a `#define` statement must precede the function. The beginning of the function obtains pointers to the S-function's continuous states, state derivatives, and first input port. The S-function uses this data to solve the equation `dx=Ax+Bu`.

```
#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);
}
```

The required `mdlTerminate` function performs any actions, such as freeing memory, necessary at the end of the simulation. In this example, the function is empty.

```
/* 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 */
}
```

The required S-function trailer includes the files necessary for simulation or code generation, as follows.

```
#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
```

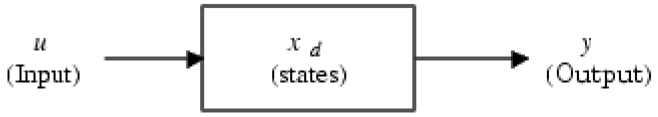
Note The `mdlOutputs` and `mdlTerminate` functions use the `UNUSED_ARG` macro to indicate that an input argument the callback requires is not used. This optional macro is defined in *`matlabroot`*`/simulink/include/simstruc_types.h`. If used, you must call this macro once for each input argument that a callback does not use.

Discrete States

The `dsfunc.c` example shows how to model a discrete system in a C MEX S-function. The following Simulink model uses this S-function.

`matlabroot``/toolbox/simulink/simdemos/simfeatures/sfcndemo_dsfunc.mdl`

Discrete systems can be modeled by the following set of equations.



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

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

The `dsfunc.c` example implements a discrete state-space equation. The `mdlOutputs` method contains the output portion and the `mdlUpdate` method contains the update portion of the discrete state-space equation. To visualize how the simulation works, see the flowchart in “How the Simulink Engine

Interacts with C S-Functions" on page 4-77. 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 Simulink engine skips the integration phase and time is moved forward to the next discrete sample hit.

matlabroot/simulink/src/dsfunc.c

The S-function begins with `#define` statements for the S-function's name and level, along with a `#include` statement for the `simstruc.h` header. After these statements, the S-function can include or define any other necessary headers, data, etc. The `dsfunc.c` example defines `U` as a pointer to the first input port's signal and initializes static variables for the state-space matrices.

```
/*  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 1990-2007 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      }
                      };
```

The required S-function method `mdlInitializeSizes` then sets up the following S-function characteristics.

- `ssSetNumSFcnParams` sets the number of expected S-function dialog parameters to zero.
- `ssGetSFcnParamsCount` determines how many parameters the user actually entered into the S-function dialog. If the number of user-specified parameters does not match the number returned by `ssGetNumSFcnParams`, the S-function errors out.
- If the S-function parameter count passes, `mdlInitializeSizes` next sets the number of continuous and discrete states using `ssSetNumContStates` and `ssSetNumDiscStates`, respectively. This example has zero continuous states and two discrete states.
- Next, the method configures the S-function to have a single input and output port, each with a width of two to match the dimensions of the state-space matrices. The method passes a value of `1` to `ssSetInputPortDirectFeedThrough` to indicate the input port has direct feedthrough.
- `ssSetNumSampleTimes` initializes one sample time, which the `mdlInitializeSampleTimes` function configures later.
- The S-function indicates that no work vectors are used by passing a value of `0` to `ssSetNumRWork`, `ssSetNumIWork`, etc. These lines could be omitted because zero is the default value for all of these macros. However, for clarity, the S-function explicitly sets the number of work vectors.
- Lastly, `ssSetOptions` sets any applicable options. In this case, the only option is `SS_OPTION_EXCEPTION_FREE_CODE`, which stipulates that the code is exception free.

The `mdlInitializeSizes` function for this example is shown below.

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

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    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 reported by the Simulink engine*/
    }

    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);
}
```

The required S-function method mdlInitializeSampleTimes specifies the S-function's sample rates. A call to ssSetSampleTime sets this S-function's first sample period to 1.0. ssSetOffsetTime then specifies an offset time of zero for the first sample rate. The call to ssSetModelReferenceSampleTimeDefaultInheritance tells the solver to use the default rule to determine if submodels containing this S-function can inherit their sample times from the parent model.

```
/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    Specifiy a sample time Of 1.0.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, 1.0);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
}
```

The optional S-function method mdlInitializeConditions initializes the discrete state vector. The #define statement before this method is required for the Simulink engine to call this function. In the example below, ssGetRealDiscStates obtains a pointer to the discrete state vector. The for loop then initializes each discrete state to one.

```
#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;
    }
}
```

The required `mdlOutputs` function computes the output signal of this S-function. The beginning of the function obtains pointers to the first output port, discrete states, and first input port. The S-function uses the data in these arrays to solve the output equation `y=Cx+Du`.

```
/* 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);
}
```

The Simulink engine calls the `mdlUpdate` function once every major integration time step to update the discrete states' values. Because this function is an optional method, a `#define` statement must precede the function. The beginning of the function obtains pointers to the S-function's discrete states and first input port. The S-function uses the data in these arrays to solve the equation `dx=Ax+Bu`, which is stored in the temporary variable `tempX` before being assigned into the discrete state vector `x`.

```
#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];
}
```

The required `mdlTerminate` function performs any actions, such as freeing memory, necessary at the end of the simulation. In this example, the function is empty.

```
/* 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 */
}
```

The required S-function trailer includes the files necessary for simulation or code generation, as follows.

```
#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
```

Note The mdlOutputs and mdlTerminate functions use the UNUSED_ARG macro to indicate that an input argument the callback requires is not used. This optional macro is defined in *matlabroot*/simulink/include/simstruc_types.h. If used, you must call this macro once for each input argument that a callback does not use.

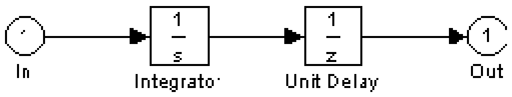
Continuous and Discrete States

The *matlabroot*/simulink/src/mixedm.c example shows a hybrid (a combination of continuous and discrete states) system. The mixedm.c example combines elements of csfunc.c and dsfunc.c. The following Simulink model uses this S-function.

matlabroot/toolbox/simulink/simdemos/simfeatures/sfcndemo_mixedm.mdl

If you have a hybrid system, the mdlDerivatives method calculates the derivatives of the continuous states of the state vector, x, and the mdlUpdate method contains the equations used to update the discrete state vector, xD. The mdlOutputs method computes the S-function outputs after checking for sample hits to determine at what point the 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.

matlabroot/simulink/src/mixedm.c

The S-function begins with #define statements for the S-function's name and level, along with a #include statement for the simstruc.h header. After these statements, the S-function can include or define any other necessary headers, data, etc. The mixedm.c example defines U as a pointer to the first input port's signal.

```
/*  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 1990-2007 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 */
```

The required S-function method `mdlInitializeSizes` then sets up the following S-function characteristics.

- `ssSetNumSFcnParams` sets the number of expected S-function dialog parameters to zero.
- `ssGetSFcnParamsCount` determines how many parameters the user actually entered into the S-function dialog. If the number of user-specified parameters does not match the number returned by `ssGetNumSFcnParams`, the S-function errors out.
- If the S-function parameter count passes, `mdlInitializeSizes` next sets the number of continuous and discrete states using `ssSetNumContStates` and `ssSetNumDiscStates`, respectively. This example has one continuous state and one discrete state.
- The S-function initializes one floating-point work vector by passing a value of `1` to `ssSetNumRWork`. No other work vectors are initialized.
- Next, the method uses `ssSetNumInputPorts` and `ssSetNumOutputPorts` to configure the S-function to have a single input and output port, each with a width of one. The method passes a value of `1` to `ssSetInputPortDirectFeedThrough` to indicate the input port has direct feedthrough.

- This S-function assigns sample times using a hybrid block-based and port-based method. The macro `ssSetNumSampleTimes` initializes two block-based sample times, which the `mdlInitializeSampleTimes` function configures later. The macros `ssSetInputPortSampleTime` and `ssSetInputPortOffsetTime` initialize the input port to have a continuous sample time with an offset of zero. Similarly, `ssSetOutputPortSampleTime` and `ssSetOutputPortOffsetTime` initialize the output port sample time to `1` with an offset of zero.
- Lastly, `ssSetOptions` sets two S-function options. `SS_OPTION_EXCEPTION_FREE_CODE` stipulates that the code is exception free and `SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED` indicates a combination of block-based and port-based sample times.

The `mdlInitializeSizes` function for this example is shown below.

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

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    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 reported by the Simulink engine*/
    }

    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 */
```

The required S-function method `mdlInitializeSampleTimes` specifies the S-function's block-based sample rates. The first call to `ssSetSampleTime` specifies that the first sample rate is continuous, with the subsequent call to `ssSetOffsetTime` setting the offset to zero. The second call to this pair of macros sets the second sample time to `1` with an offset of zero. The S-function's port-based sample times set in `mdlInitializeSizes` must all be registered as a block-based sample time. The call to `ssSetModelReferenceSampleTimeDefaultInheritance` tells the solver to use the default rule to determine if submodels containing this S-function can inherit their sample times from the parent model.

```
/* 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);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
} /* end mdlInitializeSampleTimes */
```

The optional S-function method mdlInitializeConditions initializes the continuous and discrete state vectors. The #define statement before this method is required for the Simulink engine to call this function. In this example, ssGetContStates obtains a pointer to the continuous state vector and ssGetRealDiscStates obtains a pointer to the discrete state vector. The method then sets all states' initial conditions to one.

```
#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 */
```

The required mdlOutputs function performs computations based on the current task. The macro ssIsContinuousTask checks if the continuous task is executing. If this macro returns true, ssIsSpecialSampleHit then checks if the discrete sample rate is also executing. If this macro also returns true, the method sets the value of the floating-point work vector to the current value of the continuous state, via pointers obtained using ssGetRWork and ssGetContStates, respectively. The mdlUpdate method later uses the floating-point work vector as the input to the zero-order hold. Updating the work vector in mdlOutputs ensures that the correct values are available during subsequent calls to mdlUpdate. Finally, if the S-function is running at its discrete rate, i.e., the call to ssIsSampleHit returns true, the method sets the output to the value of the discrete state.

```
/* 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 */
```

The Simulink engine calls the mdlUpdate function once every major integration time step to update the discrete states' values. Because this function is an optional method, a #define statement must precede the function. The call to ssIsSampleHit ensures the body of the method is executed only when the S-function is operating at its discrete rate. If ssIsSampleHit returns true, the method obtains pointers to the S-function's discrete state and floating-point work vector and updates the discrete state's value using the value stored in the work vector.

```
#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 */
```

The `mdlDerivatives` function calculates the continuous state derivatives. Because this function is an optional method, a `#define` statement must precede the function. The function obtains pointers to the S-function's continuous state derivative and first input port then sets the continuous state derivative equal to the value of the first input.

```
#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 */
```

The required `mdlTerminate` function performs any actions, such as freeing memory, necessary at the end of the simulation. In this example, the function is empty.

```
/* 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 */
}
```

The S-function trailer includes the files necessary for simulation or code generation, as follows.

```
#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
```

Note The `mdlUpdate` and `mdlTerminate` functions use the `UNUSED_ARG` macro to indicate that an input argument the callback requires is not used. This optional macro is defined in *`matlabroot`*`/simulink/include/simstruc_types.h`. If used, you must call this macro once for each input argument that a callback does not use.

Variable Sample Time

The example S-function `vsfunc.c` uses a variable-step sample time. The following Simulink model uses this S-function.

`matlabroot``/toolbox/simulink/simdemos/simfeatures/sfcndemo_vsfunc.mdl`

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 be used only with variable-step solvers. The `vsfunc.c` example is a discrete S-function that delays its first input by an amount of time determined by the second input.

The `vsfunc.c` example outputs 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` to calculate and set the time of the next sample hit, that is, the time when `vsfunc.c` is next called. In `mdlGetTimeOfNextVarHit`, the macro `ssGetInputPortRealSignalPtrs` gets 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

The S-function begins with `#define` statements for the S-function's name and level, along with a `#include` statement for the `simstruc.h` header. After these statements, the S-function can include or define any other necessary headers, data, etc. The `vsfunc.c` example defines `U` as a pointer to the first input port's signal.

```
/*  File    : vsfunc.c
 *  Abstract:
 *
 *      Variable step S-function example.
 *      This example S-function illustrates how to create a variable step
 *      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:
 *
 *      dt      = u(2)
 *      y(t+dt) = u(t)
 *
 *      For more details about S-functions, see simulink/src/sfuntmpl_doc.c.
 *
 *  Copyright 1990-2007 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 */
```

The required S-function method `mdlInitializeSizes` then sets up the following S-function characteristics.

- `ssSetNumSFcnParams` sets the number of expected S-function dialog parameters to zero.
- `ssGetSFcnParamsCount` determines how many parameters the user actually entered into the S-function dialog. If the number of user-specified parameters does not match the number returned by `ssGetNumSFcnParams`, the S-function errors out.
- If the S-function parameter count passes, `mdlInitializeSizes` next sets the number of continuous and discrete states using `ssSetNumContStates` and `ssSetNumDiscStates`, respectively. This example has no continuous states and one discrete state.
- Next, the method uses `ssSetNumInputPorts` and `ssSetNumOutputPorts` to configure the S-function to have a single input and output port. Calls to `ssSetInputPortWidth` and `ssSetOutputPortWidth` assign widths to these input and output ports. The method passes a value of `1` to `ssSetInputPortDirectFeedThrough` to indicate the input port has direct feedthrough.
- `ssSetNumSampleTimes` then initializes one sample time, which the `mdlInitializeSampleTimes` function configures later.
- The S-function indicates that no work vectors are used by passing a value of `0` to `ssSetNumRWork`, `ssSetNumIWork`, etc. These lines could be omitted because zero is the default value for all of these macros. However, for clarity, the S-function explicitly sets the number of work vectors.
- Next, `ssGetSimMode` checks if the S-function is being run in a simulation or by the Real-Time Workshop product. If `ssGetSimMode` returns `SS_SIMMODE_RTWGEN` and `ssIsVariableStepSolver` returns `false`, indicating use with the Real-Time Workshop product and a fixed-step solver, then the S-function errors out.
- Lastly, `ssSetOptions` sets any applicable options. In this case, the only option is `SS_OPTION_EXCEPTION_FREE_CODE`, which stipulates that the code is exception free.

The `mdlInitializeSizes` function for this example is shown below.

```
/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    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 reported by the Simulink engine*/
    }

    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);
}
```

The required S-function method `mdlInitializeSampleTimes` specifies the S-function's sample rates. The input argument `VARIABLE_SAMPLE_TIME` passed to `ssSetSampleTime` specifies that this S-function has a variable-step sample time and `ssSetOffsetTime` specifies an offset time of zero. The call to `ssSetModelReferenceSampleTimeDefaultInheritance` tells the solver to use the default rule to determine if submodels containing this S-function can inherit their sample times from the parent model. Because the S-function has a variable-step sample time, vsfunc.c must calculate the time of the next sample hit in the mdlGetTimeOfNextVarHit method, shown later.

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

The optional S-function method mdlInitializeConditions initializes the discrete state vector. The #define statement before this method is required for the Simulink engine to call this function. In the example, the method uses ssGetRealDiscStates to obtain a pointer to the discrete state vector and sets the state's initial value to zero.

```
#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;
}
```

The optional mdlGetTimeOfNextVarHit method calculates the time of the next sample hit. Because this method is optional, a #define statement precedes it. First, this method obtains a pointer to the first input port's signal using ssGetInputPortRealSignalPtrs. If the input signal's second element is positive, the macro ssGetT gets the simulation time t. The macro ssSetTNext sets the time of the next hit equal to `t+(*U[1])`, delaying the output by the amount of time specified by the input's second element (`*U[1]`).

```
#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));
}
```

The required `mdlOutputs` function computes the S-function's output signal. The function obtains pointers to the first output port and discrete state and then assigns the state's current value to the output.

```
/* 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];
}
```

The `mdlUpdate` function updates the discrete state's value. Because this method is optional, a `#define` statement precedes it. The function first obtains pointers to the S-function's discrete state and first input port then assigns the value of the first element of the first input port signal to the state.

```
#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);
}
```

The required `mdlTerminate` function performs any actions, such as freeing memory, necessary at the end of the simulation. In this example, the function is empty.

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

The required S-function trailer includes the files necessary for simulation or code generation, as follows.

```
#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
```

Array Inputs and Outputs

The example S-function *`matlabroot`*`/simulink/src/sfun_matadd.c` demonstrates how to implement a matrix addition block. The following Simulink model uses this S-function.

matlabroot/toolbox/simulink/simdemos/simfeatures/sfcndemo_matadd.mdl

The S-function adds signals of various dimensions to a parameter value entered in the S-function. The S-function accepts and outputs 2-D or n-D signals.

matlabroot/simulink/src/sfun_matadd.c

The S-function begins with `#define` statements for the S-function's name and level, along with a `#include` statement for the `simstruc.h` header. After these statements, the S-function includes or defines any other necessary headers, data, etc. This example defines additional variables for the number of S-function parameters, the S-function parameter value, and the flag `EDIT_OK` that indicates if the parameter value can be edited during simulation.

```
/* SFUN_MATADD matrix support example.
 *   C MEX S-function for matrix addition with one input port,
 *   one output port, and one parameter.
 *
 *  Input Signal:  2-D or n-D array
 *  Parameter:     2-D or n-D array
 *  Output Signal: 2-D or n-D array
 *
 *  Input   parameter    output
 *  --------------------------------
 *  scalar   scalar      scalar
 *  scalar   matrix     matrix     (input scalar expansion)
 *  matrix   scalar     matrix     (parameter scalar expansion)
 *  matrix   matrix     matrix
 *
 *  Copyright 1990-2007 The MathWorks, Inc.
 */
#define S_FUNCTION_NAME  sfun_matadd
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

enum {PARAM = 0, NUM_PARAMS};

#define PARAM_ARG ssGetSFcnParam(S, PARAM)
```

```
#define EDIT_OK(S, ARG) \
  (!((ssGetSimMode(S) == SS_SIMMODE_SIZES_CALL_ONLY) \
  && mxIsEmpty(ARG)))
```

The S-function next implements the `mdlCheckParameters` method to validate the S-function dialog parameters. The `#ifdef` statement checks that the S-function is compiled as a MEX-file, instead of for use with the Real-Time Workshop product. Because `mdlCheckParameters` is optional, the S-function code contains a `#define` statement to register the method. The body of the function checks that the S-function parameter value is not empty. If the parameter check fails, the S-function errors out with a call to `ssSetErrorStatus`.

```
#ifdef MATLAB_MEX_FILE
#define MDL_CHECK_PARAMETERS
/* Function: mdlCheckParameters ================================

 * Abstract:
 *    Verify parameter settings.
 */
static void mdlCheckParameters(SimStruct *S)
{
    if(EDIT_OK(S, PARAM_ARG)){
        /* Check that parameter value is not empty*/
        if( mxIsEmpty(PARAM_ARG) ) {
          ssSetErrorStatus(S, "Invalid parameter specified. The"
                              "parameter must be non-empty");
          return;
        }
    }
} /* end mdlCheckParameters */
#endif
```

The required S-function method `mdlInitializeSizes` then sets up the following S-function characteristics.

- `ssSetNumSFcnParams` sets the number of expected S-function dialog parameters to one, as defined by the variable `NUM_PARAMS`.

- If this S-function is compiled as a MEX-file, `ssGetSFcnParamsCount` determines how many parameters the user actually entered into the S-function dialog. If the number of user-specified parameters matches the number returned by `ssGetNumSFcnParams`, the method calls `mdlCheckParameters` to validate the user-entered data. Otherwise, the S-function errors out.
- If the parameter check passes, the S-function specifies that all S-function parameters are tunable using `ssSetSFcnParamTunable`.
- The S-function then invokes `ssAllowSignalsWithMoreThan2D` to allow the S-function to accept n-D signals.
- Next, `ssSetNumOutputPorts` and `ssSetNumInputPorts` specify that the S-function has a single output port and a single input port.
- The S-function uses `ssSetInputPortDimensionInfo` to specify that the input port is dynamically sized. In this case, the S-function needs to implement an `mdlSetInputPortDimensionInfo` method to set the actual input dimension.
- The output dimensions depend on the dimensions of the S-function parameter. If the parameter is a scalar, the call to `ssSetOutputPortDimensionInfo` specifies that the output port dimensions are dynamically sized. If the parameter is a matrix, the output port dimensions are initialized to the dimensions of the S-function parameter. In this case, the macro `DECL_AND_INIT_DIMSINFO` initializes a `dimsInfo` structure. The S-function assigns the width, size, and dimensions of the S-function parameter into the `dimsInfo` structure and then passes this structure to `ssSetOutputPortDimensionInfo` to set the output port dimensions accordingly.
- The S-function specifies that the input port has direct feedthrough by passing a value of `1` to `ssSetInputPortDirectFeedThrough`.
- `ssSetNumSampleTimes` initializes one sample time, to be configured later in the `mdlInitializeSampleTimes` method.
- Lastly, `ssSetOptions` sets any applicable options. In this case, `SS_OPTION_EXCEPTION_FREE_CODE` stipulates that the code is exception free and `SS_OPTION_WORKS_WITH_CODE_REUSE` signifies that this S-function is compatible with the subsystem code reuse feature of the Real-Time Workshop product.

```
/* Function: mdlInitializeSizes ================================
 * Abstract:
 *   Initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, NUM_PARAMS);

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

   {
      int iParam = 0;
      int nParam = ssGetNumSFcnParams(S);

      for ( iParam = 0; iParam < nParam; iParam++ )
        {
          ssSetSFcnParamTunable( S, iParam, SS_PRM_TUNABLE );
        }
    }

   /* Allow signal dimensions greater than 2 */
   ssAllowSignalsWithMoreThan2D(S);

   /* Set number of input and output ports */
   if (!ssSetNumInputPorts( S,1)) return;
   if (!ssSetNumOutputPorts(S,1)) return;

   /* Set dimensions of input and output ports */
   {
      int_T pWidth = mxGetNumberOfElements(PARAM_ARG);
      /* Input can be a scalar or a matrix signal. */
      if(!ssSetInputPortDimensionInfo(S,0,DYNAMIC_DIMENSION)) {
          return; }

      if( pWidth == 1) {
        /* Scalar parameter: output dimensions are unknown. */
        if(!ssSetOutputPortDimensionInfo(S,0,DYNAMIC_DIMENSION)){
            return; }
        }
      else{
         /*
          * Non-scalar parameter: output dimensions are the same
          * as the parameter dimensions. To support n-D signals,
          * must use a dimsInfo structure to specify dimensions.
          */
          DECL_AND_INIT_DIMSINFO(di); /*Initializes structure*/
          int_T      pSize = mxGetNumberOfDimensions(PARAM_ARG);
          const int_T *pDims = mxGetDimensions(PARAM_ARG);
          di.width   = pWidth;
          di.numDims = pSize;
          di.dims    = pDims;
          if(!ssSetOutputPortDimensionInfo(S, 0, &di)) return;
          }
    }
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    ssSetNumSampleTimes(S, 1);
    ssSetOptions(S,
                 SS_OPTION_WORKS_WITH_CODE_REUSE |
                 SS_OPTION_EXCEPTION_FREE_CODE);
} /* end mdlInitializeSizes */
```

The required S-function method `mdlInitializeSampleTimes` specifies the S-function's sample rates. To specify that this S-function inherits its sample time from its driving block, the S-function calls `ssSetSampleTime` with the input argument `INHERITED_SAMPLE_TIME`. The call to `ssSetModelReferenceSampleTimeDefaultInheritance` tells the solver to use the default rule to determine if submodels containing this S-function can inherit their sample times from the parent model.

```
/* Function: mdlInitializeSampleTimes ==========================
 * Abstract:
 *    Initialize the sample times array.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
```

```
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
} /* end mdlInitializeSampleTimes */
```

The S-function calls the `mdlSetWorkWidths` method to register its run-time parameters. Because `mdlSetWorkWidths` is an optional method, a `#define` statement precedes it. The method first initializes a name for the run-time parameter and then uses `ssRegAllTunableParamsAsRunTimeParams` to register the run-time parameter.

```
/* Function: mdlSetWorkWidths ==================================
 * Abstract:
 *    Set up run-time parameter.
 */
#define MDL_SET_WORK_WIDTHS
static void mdlSetWorkWidths(SimStruct *S)
{
    const char_T    *rtParamNames[] = {"Operand"};
    ssRegAllTunableParamsAsRunTimeParams(S, rtParamNames);
} /* end mdlSetWorkWidths */
```

The S-function's `mdlOutputs` method uses a `for` loop to calculate the output as the sum of the input and S-function parameter. The S-function handles n-D arrays of data using a single index into the array.

```
/* Function: mdlOutputs ========================================
 * Abstract:
 *   Compute the outputs of the S-function.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtr = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y   = ssGetOutputPortRealSignal(S,0);
    const real_T      *p   = mxGetPr(PARAM_ARG);

    int_T             uWidth = ssGetInputPortWidth(S,0);
    int_T             pWidth = mxGetNumberOfElements(PARAM_ARG);
    int_T             yWidth = ssGetOutputPortWidth(S,0);
    int               i;
```

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

        /*
         * Note1: Matrix signals are stored in column major order.
         * Note2: Access each matrix element by one index not two
         *        indices. For example, if the output signal is a
         *        [2x2] matrix signal,
         *        -          -
         *       | y[0]  y[2] |
         *       | y[1]  y[3] |
         *       -           -
         *       Output elements are stored as follows:
         *           y[0] --> row = 0, col = 0
         *           y[1] --> row = 1, col = 0
         *           y[2] --> row = 0, col = 1
         *           y[3] --> row = 1, col = 1
         */

        for (i = 0; i < yWidth; i++) {
            int_T uIdx = (uWidth == 1) ? 0 : i;
            int_T pIdx = (pWidth == 1) ? 0 : i;

            y[i] = *uPtr[uIdx] + p[pIdx];
        }
    } /* end mdlOutputs */
```

During signal propagation, the S-function calls the optional `mdlSetInputPortDimensionInfo` method with the candidate input port dimensions stored in `dimsInfo`. The `#if defined` statement checks that the S-function is compiled as a MEX-file. Because `mdlSetInputPortDimensionInfo` is an optional method, a `#define` statement precedes it. In `mdlSetInputPortDimensionInfo`, the S-function uses `ssSetInputPortDimensionInfo` to set the dimensions of the input port to the candidate dimensions. If the call to this macro succeeds, the S-function further checks the candidate dimensions to ensure that the input signal is either a 2-D scalar or a matrix. If this condition is met and the output port dimensions are still dynamically sized, the S-function calls `ssSetOutputPortDimensionInfo` to set the dimension of the output port to the same candidate dimensions.

The ssSetOutputPortDimensionInfo macro cannot modify the output port dimensions if they are already specified.

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_DIMENSION_INFO
/* Function: mdlSetInputPortDimensionInfo ======================
 * Abstract:
 *    This routine is called with the candidate dimensions for
 *    an input port with unknown dimensions. If the proposed
 *    dimensions are acceptable, the routine should go ahead and
 *    set the actual port dimensions. If they are unacceptable
 *    an error should be generated via ssSetErrorStatus.
 *    Note that any other input or output ports whose dimensions
 *    are implicitly defined by virtue of knowing the dimensions
 *    of the given port can also have their dimensions set.
 */
static void mdlSetInputPortDimensionInfo(SimStruct        *S,
   int_T            port,
   const DimsInfo_T *dimsInfo)
{
    int_T  pWidth          = mxGetNumberOfElements(PARAM_ARG);
    int_T  pSize           = mxGetNumberOfDimensions(PARAM_ARG);
    const int_T  *pDims    = mxGetDimensions(PARAM_ARG);

    int_T  uNumDims = dimsInfo->numDims;
    int_T  uWidth   = dimsInfo->width;
    int_T  *uDims   = dimsInfo->dims;

    int_T numDims;
    boolean_T  isOk = true;
    int iParam = 0;
    int_T outWidth = ssGetOutputPortWidth(S, 0);

    /* Set input port dimension */
    if(!ssSetInputPortDimensionInfo(S, port, dimsInfo)) return;

    /*
     * The block only accepts 2-D or higher signals. Check
     * number of dimensions. If the parameter and the input
     * signal are non-scalar, their dimensions must be the same.
```

```
      */
     isOk = (uNumDims >= 2) && (pWidth == 1 || uWidth == 1 ||
        pWidth == uWidth);
     numDims = (pSize != uNumDims) ? numDims : uNumDims;

     if(isOk && pWidth > 1 && uWidth > 1){
         for ( iParam = 0; iParam < numDims; iParam++ ) {
             isOk = (pDims[iParam] == uDims[iParam]);
             if(!isOk) break;
         }
     }

     if(!isOk){
         ssSetErrorStatus(S,"Invalid input port dimensions. The "
         "input signal must be a 2-D scalar signal, or it must "
         "be a matrix with the same dimensions as the parameter "
         "dimensions.");
         return;
     }

     /* Set the output port dimensions */
     if (outWidth == DYNAMICALLY_SIZED){
       if(!ssSetOutputPortDimensionInfo(S,port,dimsInfo)) return;
     }
 } /* end mdlSetInputPortDimensionInfo */
```

During signal propagation, if any output ports have unknown dimensions, the S-function calls the optional `mdlSetOutputPortDimensionInfo` method. Because this method is optional, a `#define` statement precedes it. In `mdlSetOutputPortDimensionInfo`, the S-function uses `ssSetOutputPortDimensionInfo` to set the dimensions of the output port to the candidate dimensions `dimsInfo`. If the call to this macro succeeds, the S-function further checks the candidate dimensions to ensure that the input signal is either a 2-D or n-D matrix. If this condition is not met, the S-function errors out with a call to `ssSetErrorStatus`. Otherwise, the S-function calls `ssSetInputPortDimensionInfo` to set the dimension of the input port to the same candidate dimensions.

```
# define MDL_SET_OUTPUT_PORT_DIMENSION_INFO
/* Function: mdlSetOutputPortDimensionInfo =====================
 * Abstract:
 *    This routine is called with the candidate dimensions for
 *    an output port with unknown dimensions. If the proposed
 *    dimensions are acceptable, the routine should go ahead and
 *    set the actual port dimensions. If they are unacceptable
 *    an error should be generated via ssSetErrorStatus.
 *    Note that any other input or output ports whose dimensions
 *    are implicitly defined by virtue of knowing the dimensions
 *    of the given port can also have their dimensions set.
 */
static void mdlSetOutputPortDimensionInfo(SimStruct        *S,
   int_T            port,
   const DimsInfo_T *dimsInfo)
{
    /*
     * If the block has scalar parameter, the output dimensions
     * are unknown. Set the input and output port to have the
     * same dimensions.
     */
    if(!ssSetOutputPortDimensionInfo(S, port, dimsInfo)) return;

    /* The block only accepts 2-D or n-D signals.
     * Check number of dimensions.
     */
    if (!(dimsInfo->numDims >= 2)){
        ssSetErrorStatus(S, "Invalid output port dimensions. "
        "The output signal must be a 2-D or n-D array (matrix) "
        "signal.");
        return;
    }else{
       /* Set the input port dimensions */
       if(!ssSetInputPortDimensionInfo(S,port,dimsInfo)) return;
    }
} /* end mdlSetOutputPortDimensionInfo */
```

Because the S-function has ports that are dynamically sized, it must provide an `mdlSetDefaultPortDimensionInfo` method. The Simulink engine invokes this method during signal propagation when it cannot determine the dimensionality of the signal connected to the block's input port. This situation can happen, for example, if the input port is unconnected. In this example, the `mdlSetDefaultPortDimensionInfo` method sets the input and output ports dimensions to a scalar.

```
# define MDL_SET_DEFAULT_PORT_DIMENSION_INFO
/* Function: mdlSetDefaultPortDimensionInfo ====================
 *    This routine is called when the Simulink engine is not able
 *    to find dimension candidates for ports with unknown dimensions.
 *    This function must set the dimensions of all ports with
 *    unknown dimensions.
 */
static void mdlSetDefaultPortDimensionInfo(SimStruct *S)
{
    int_T outWidth = ssGetOutputPortWidth(S, 0);
    /* Input port dimension must be unknown. Set it to scalar.*/
    if(!ssSetInputPortMatrixDimensions(S, 0, 1, 1)) return;
    if(outWidth == DYNAMICALLY_SIZED){
        /* Output dimensions are unknown. Set it to scalar. */
        if(!ssSetOutputPortMatrixDimensions(S, 0, 1, 1)) return;
    }
} /* end mdlSetDefaultPortDimensionInfo */
#endif
```

The required `mdlTerminate` function performs any actions, such as freeing memory, necessary at the end of the simulation. In this example, the function is empty.

```
/* Function: mdlTerminate ======================================
 * Abstract:
 *    Called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
    UNUSED_ARG(S); /* unused input argument */

} /* end mdlTerminate */
```

The required S-function's trailer includes the files necessary for simulation or code generation.

```
#ifdef MATLAB_MEX_FILE
  #include "simulink.c"
```

```
#else
  #include "cg_sfun.h"
#endif

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

Note The `mdlOutputs` and `mdlTerminate` functions use the `UNUSED_ARG` macro to indicate that an input argument the callback requires is not used. This optional macro is defined in *`matlabroot`*`/simulink/include/simstruc_types.h`. You must call this macro once for each input argument that a callback does not use.

Zero-Crossing Detection

The example S-function *`matlabroot`*`/simulink/src/sfun_zc_sat.c` demonstrates how to implement a Saturation block. The following Simulink model uses this S-function.

`sfcndemo_sfun_zc_sat.mdl`

The S-function works with either fixed-step or variable-step solvers. When this S-function inherits a continuous sample time and uses a variable-step solver, it uses a zero-crossings algorithm to locate the exact points at which the saturation occurs.

matlabroot/simulink/src/sfun_zc_sat.c

The S-function begins with `#define` statements for the S-function's name and level, along with a `#include` statement for the `simstruc.h` header. After these statements, the S-function includes or defines any other necessary headers, data, etc. This example defines various parameters associated with the upper and lower saturation bounds.

```
/*  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 1990-2007 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) )
```

This S-function next implements the `mdlCheckParameters` method to check the validity of the S-function dialog parameters. Because this method is optional, a `#define` statement precedes it. The `#if defined` statement checks that this function is compiled as a MEX-file, instead of for use with the Real-Time Workshop product. The body of the function performs basic checks to ensure that the user entered real vectors of equal length for the upper and lower saturation limits. If the parameter checks fail, the S-function errors out.

```
#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 */
```

The required S-function method `mdlInitializeSizes` sets up the following S-function characteristics.

- `ssSetNumSFcnParams` sets the number of expected S-function dialog parameters to two, as defined previously in the variable `N_PAR`.
- If this method is compiled as a MEX-file, `ssGetSFcnParamsCount` determines how many parameters the user actually entered into the S-function dialog. If the number of user-specified parameters matches the number returned by `ssGetNumSFcnParams`, the method calls `mdlCheckParameters` to check the validity of the user-entered data. Otherwise, the S-function errors out.
- If the parameter check passes, the S-function determines the maximum number of elements entered into either the upper or lower saturation limit parameter. This number is needed later to determine the appropriate output width.

- Next, the number of continuous and discrete states is set using ssSetNumContStates and ssSetNumDiscStates, respectively. This example has no continuous or discrete states.
- The method specifies that the S-function has a single output port using ssSetNumOutputPorts and sets the width of this output port using ssSetOutputPortWidth. The output port width is either the maximum number of elements in the upper or lower saturation limit or is dynamically sized. Similar code specifies a single input port and indicates the input port has direct feedthrough by passing a value of 1 to ssSetInputPortDirectFeedThrough.
- ssSetNumSampleTimes initializes one sample time, which the mdlInitializeSampleTimes function configures later.
- The S-function indicates that no work vectors are used by passing a value of 0 to ssSetNumRWork, ssSetNumIWork, etc. These lines could be omitted because zero is the default value for all of these macros. However, for clarity, the S-function explicitly sets the number of work vectors.
- The method initializes the zero-crossing detection work vectors using ssSetNumModes and ssSetNumNonsampledZCs. The mdlSetWorkWidths method specifies the length of these dynamically sized vectors later.
- Lastly, ssSetOptions sets any applicable options. In this case, SS_OPTION_EXCEPTION_FREE_CODE stipulates that the code is exception free and SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION permits scalar expansion of the input without having to provide an mdlSetInputPortWidth function.

The mdlInitializeSizes function for this example is shown below.

```
/* 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 reported by the Simulink engine*/
    }
#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 */
```

The required S-function method `mdlInitializeSampleTimes` specifies the S-function's sample rates. The input argument `INHERITED_SAMPLE_TIME` passed to `ssSetSampleTime` specifies that this S-function inherits its sample time from its driving block. The call to `ssSetModelReferenceSampleTimeDefaultInheritance` tells the solver to use the default rule to determine if submodels containing this S-function can inherit their sample times from the parent model.

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

The optional method `mdlSetWorkWidths` initializes the size of the zero-crossing detection work vectors. Because this method is optional, a `#define` statement precedes it. The `#if defined` statement checks that the S-function is being compiled as a MEX-file. Zero-crossing detection can be done only when the S-function is running at a continuous sample rate using a variable-step solver. The `if` statement uses `ssIsVariableStepSolver`, `ssGetSampleTime`, and `ssGetOffsetTime` to determine if this condition is met. If so, the method sets the number of modes equal to the width of the first output port and the number of nonsampled zero crossings to twice this amount. Otherwise, the method sets both values to zero.

```
#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 */
```

After declaring variables for the input and output signals, the `mdlOutputs` functions uses an `if-else` statement to create blocks of code used to calculate the output signal based on whether the S-function uses a fixed-step or variable-step solver. The `if` statement queries the length of the nonsampled zero-crossing vector. If the length, set in `mdlWorkWidths`, is zero, then no zero-crossing detection is done and the output signals are calculated directly from the input signals. Otherwise, the function uses the mode work vector to determine how to calculate the output signal. If the simulation is at a major time step, i.e., `ssIsMajorTimeStep` returns `true`, `mdlOutputs` determines which mode the simulation is running in, either saturated at the upper limit, saturated at the lower limit, or not saturated. Then, for both major and minor time steps, the function calculates an output based on this mode. If the mode changed between the previous and current time step, then a zero crossing occurred. The `mdlZeroCrossings` function, not `mdlOutputs`, indicates this crossing to the solver.

```
/* 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 */
```

The `mdlZeroCrossings` method determines if a zero crossing occurred between the previous and current time step. The method obtains a pointer to the input signal using `ssGetInputPortRealSignalPtrs`. A comparison of this signal's value to the value of the upper and lower saturation limits determines values for the elements of the nonsampled zero-crossing vector. If any element of the nonsampled zero-crossing vector switches from negative to positive, or positive to negative, a zero crossing occurred. In the event of a zero crossing, the Simulink engine modifies the step size and recalculates the outputs to try to locate the exact zero crossing.

```
#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 */
```

The S-function concludes with the required `mdlTerminate` function. In this example, the function is empty.

```
    /* 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 */
}
```

The required S-function trailer includes the files necessary for simulation or code generation, as follows.

```
#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
```

Note The `mdlOutputs` and `mdlTerminate` functions use the `UNUSED_ARG` macro to indicate that an input argument the callback requires is not used. This optional macro is defined in `*matlabroot*/simulink/include/simstruc_types.h`. If used, you must call this macro once for each input argument that a callback does not use.

Discontinuities in Continuous States

The example S-function `stvctf.c` demonstrates a time-varying continuous transfer function. The following Simulink model uses this S-function.

`sfcndemo_stvctf.mdl`

The S-function 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

The S-function begins with `#define` statements for the S-function's name and level, along with a `#include` statement for the `simstruc.h` header. After these statements, the S-function includes or defines any other necessary headers, data, etc. This example defines parameters for the transfer function's numerator and denominator, which are entered into the S-function's dialog. The comments at the beginning of this S-function provide additional information on the purpose of the work vectors in this example.

```
/*
 * 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
 *   Configuraion 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 1990-7 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
```

This S-function implements the `mdlCheckParameters` method to check the validity of the S-function dialog parameters. Because this method is optional, a #define statement precedes it. The #if defined statement checks that this function is compiled as a MEX-file, instead of for use with the Real-Time Workshop product. The body of the function performs basic checks to ensure that the user entered real vectors for the numerator and denominator, and that the denominator has a higher order than the numerator. If the parameter check fails, the S-function errors out.

```
#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 */
```

The required S-function method `mdlInitializeSizes` then sets up the following S-function characteristics.

- `ssSetNumSFcnParams` sets the number of expected S-function dialog parameters to three, as defined previously in the variable `NPARAMS`.
- If this method is compiled as a MEX-file, `ssGetSFcnParamsCount` determines how many parameters the user entered into the S-function dialog. If the number of user-specified parameters matches the number returned by `ssGetNumSFcnParams`, the method calls `mdlCheckParameters` to check the validity of the user-entered data. Otherwise, the S-function errors out.
- If the parameter check passes, the S-function specifies the number of continuous and discrete states using `ssSetNumContStates` and `ssSetNumDiscStates`, respectively. This example has no discrete states and sets the number of continuous states based on the number of coefficients in the transfer function's denominator.
- Next, `ssSetNumInputPorts` specifies that the S-function has a single input port and sets its width to one plus twice the length of the denominator using `ssSetInputPortWidth`. The method uses the value provided by the third S-function dialog parameter as the input port's sample time. This parameter indicates the rate at which the transfer function is modified during simulation. The S-function specifies that the input port has direct feedthrough by passing a value of `1` to `ssSetInputPortDirectFeedThrough`.

- ssSetNumOutputPorts specifies that the S-function has a single output port. The method uses ssSetOutputPortWidth to set the width of this output port, ssSetOutputPortSampleTime to specify that the output port has a continuous sample time, and ssSetOutputPortOffsetTime to set the offset time to zero.
- ssSetNumSampleTimes then initializes two sample times, which the mdlInitializeSampleTimes function configures later.
- The method passes a value of four times the number of denominator coefficients to ssSetNumRWork to set the length of the floating-point work vector. ssSetNumIWork then sets the length of the integer work vector to two. The RWork vectors store two banks of transfer function coefficients, while the IWork vector indicates which bank in the RWork vector is currently in use. The S-function sets the length of all other work vectors to zero. These lines could be omitted because zero is the default value for these macros. However, for clarity, the S-function explicitly sets the number of work vectors.
- Lastly, ssSetOptions sets any applicable options. In this case, SS_OPTION_EXCEPTION_FREE_CODE stipulates that the code is exception free.

The mdlInitializeSizes function for this example is shown below.

```
/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    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 reported by the Simulink engine*/
        }
#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 */
```

The required S-function method `mdlInitializeSampleTimes` specifies the S-function's sample rates. The first call to `ssSetSampleTime` specifies that the first sample rate is continuous and the subsequent call to `ssSetOffsetTime` sets the offset to zero. The second call to this pair of macros sets the second sample time to the value of the third S-function parameter with an offset of zero. The call to `ssSetModelReferenceSampleTimeDefaultInheritance` tells the solver to use the default rule to determine if submodels containing this S-function can inherit their sample times from the parent model.

```
    /* 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);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);


} /* end mdlInitializeSampleTimes */
```

The optional S-function method `mdlInitializeConditions` initializes the continuous state vector and the initial numerator and denominator vectors. The `#define` statement before this method is required for the Simulink engine to call this function. The function initializes the continuous states to zero. The numerator and denominator coefficients are initialized from the first two S-function parameters, normalized by the first denominator coefficient. The function sets the value stored in the IWork vector to zero, to indicate that the first bank of numerator and denominator coefficients stored in the RWork vector is currently in use.

```
#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 */
```

The `mdlOutputs` function calculates the S-function output signals when the S-function is simulating in a continuous task, i.e., `ssIsContinuousTask` is `true`. If the simulation is also at a major time step, `mdlOutputs` checks if the numerator and denominator coefficients need to be updated, as indicated by a switch in the active bank stored in the IWork vector. At both major and minor time steps, the S-function calculates the output using the numerator coefficients stored in the active bank.

```
/* 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 */
```

Although this example has no discrete states, the method still implements the `mdlUpdate` function to update the transfer function coefficients at every major time step. Because this method is optional, a `#define` statement precedes it. The method uses `ssGetInputPortRealSignalPtrs` to obtain a pointer to the input signal. The input signal's values become the new transfer function coefficients, which the S-function stores in the bank of the inactive RWork vector. When the `mdlOutputs` function is later called at this major time step, it updates the active bank to be this updated bank of coefficients.

```
#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)
{
    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 */
```

The `mdlDerivatives` function calculates the continuous state derivatives. The function uses the coefficients from the active bank to solve a controllable state-space representation of the transfer function.

```
#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 */
```

The required `mdlTerminate` function performs any actions, such as freeing memory, necessary at the end of the simulation. In this example, the function is empty.

```
    /* 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 */
```

The required S-function trailer includes the files necessary for simulation or code generation, as follows.

```
#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
```

Note The `mdlTerminate` function uses the `UNUSED_ARG` macro to indicate that an input argument the callback requires is not used. This optional macro is defined in *`matlabroot`*`/simulink/include/simstruc_types.h`. If used, you must call this macro once for each input argument that a callback does not use.

S-Function Callback Methods — Alphabetical List

Every user-written S-function must implement a set of methods, called *callback methods* or simply *callbacks*, that the Simulink engine invokes when simulating a model that contains the S-function. Some callback methods are optional. The engine 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. For a list of required callback methods, see "Callback Methods That an S-Function Must Implement" on page 4-52.

mdlCheckParameters

Purpose

Check the validity of an S-function's parameters

Required

No

C Syntax

```
#define MDL_CHECK_PARAMETERS
void mdlCheckParameters(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
CheckParameters(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing a Level-2 M-File S-Function block.

Description

Verifies new parameter settings whenever parameters change or are reevaluated during a simulation. For C MEX S-functions, this method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement to be compatible with code generation targets that support non-inlined S-functions.

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, the Simulink engine 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`.

C Example

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

Note Since `mdlCheckParameters` is an optional method, a `#define MDL_CHECK_PARAMETERS` statement precedes the function. Also, since the Real-Time Workshop product does not support code generation for `mdlCheckParameters`, the function is wrapped in a `#if defined(MATLAB_MEX_FILE)` statement.

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

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

```
static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumSFcnParams(S, 1);  /* Number of expected parameters */
#if defined(MATLAB_MEX_FILE)
   if(ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S) {
     mdlCheckParameters(S);
     if(ssGetErrorStatus(S) != NULL) return;
   } else {
     return; /* The Simulink engine reports a mismatch error. */
   }
#endif
 ...
}
```

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

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

M Example

In a Level-2 M-file S-function, the `setup` method registers the `CheckParameters` method as follows

```
s.RegBlockMethod('CheckParameters', @CheckParam);
```

The subfunction `CheckParam` then verifies the S-function parameters. In this example, the function checks that the second parameter, an upper limit value, is greater than the first S-function parameter, a lower limit value.

```
function CheckParam(s)

% Check that upper limit is greater than lower limit
```

```
lowerLim = s.DialogPrm(1).Data;
upperLim = s.DialogPrm(2).Data;

if upperLim <= lowerLim,
   error('The upper limit must be greater than the lower limit.');
end
```

Languages

C, C++, M

See Also

`mdlProcessParameters`, `ssGetSFcnParamsCount`

Purpose

Compute the S-function's derivatives

Required

No

C Syntax

```
#define MDL_DERIVATIVES
void mdlDerivatives(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
Derivatives(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block

Description

The Simulink engine 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. In a C MEX S-function, use `ssGetdX` to get a pointer to the derivatives vector. In a Level-2 M-file S-function, use the run-time object's `Derivatives` method.

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.

Note When generating code for a noninlined C MEX S-function that contains this method, make sure the method is not wrapped in a `#if defined(MATLAB_MEX_FILE)` statement. For example:

```
#define MDL_DERIVATIVES
#if defined(MDL_DERIVATIVES) && defined(MATLAB_MEX_FILE)
static void mdlDerivatives(SimStruct *S)
{
   /* Add mdlDerivatives code here *
}
#endif
```

The `define` statement makes the `mdlDerivatives` method available only to a MATLAB MEX-file. If the S-function is not inlined, the Real-Time Workshop product cannot use this method, resulting in link or run-time errors.

C Example

For a C MEX S-function example, see `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/csfunc.c`.

M Example

For a Level-2 M-file S-function example, see `*matlabroot*/toolbox/simulink/simdemos/simfeatures/msfcn_limintm.m`.

Languages

C, C++, M

See Also

`ssGetdx`

Purpose Respond to disabling of an enabled system containing this block

Required No

C Syntax

```
#define MDL_DISABLE
void mdlDisable(SimStruct *S)
```

C Arguments

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

M Syntax

```
Disable(s)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description The Simulink engine invokes this optional method if this block resides in an enabled subsystem and the enabled subsystem changes from an enabled to a disabled state at the current time step. Your S-function can use this method to perform any actions required by the disabling of the containing subsystem.

Languages C, C++, M

See Also `mdlEnable`

mdlEnable

Purpose Respond to enabling of an enabled system containing this block

Required No

C Syntax

```
#define MDL_ENABLE
void mdlEnable(SimStruct *S)
```

C Arguments

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

M Syntax

```
Enable(s)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description The Simulink engine invokes this optional method if this block resides in an enabled subsystem and the enabled subsystem changes from a disabled to an enabled state at the current time step. Your S-function can use this method to perform any actions required by the enabling of the containing subsystem.

Languages C, C++, M

See Also `mdlDisable`

Purpose

Return the S-function simulation state as a valid MATLAB data structure, such as a matrix structure or a cell array.

Required

No

C Syntax

```
#define MDL_SIM_STATE
mxArray* mdlSetSimState(SimStruct* S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
GetSimState(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

The Simulink engine invokes this custom method to get the simulation state (SimState) of the model containing `S`. A call to this method should occur after `mdlStart` and before `mdlTerminate` to ensure that all of the S-function data structures (e.g., states, DWork vectors, and outputs) are available.

C Example

```
% Function: mdlGetSimState
% Abstract:
% Package the RunTimeData structure as a MATLAB structure
% and return it.
%
static mxArray* mdlGetSimState(SimStruct* S)
{
    int n = ssGetInputPortWidth(S, 0);
    RunTimeData_T* rtd =
  (RunTimeData_T*)ssGetPWorkValue(S, 0);
```

```
    /* Create a MATLAB structure to hold the run-time data */
    mxArray* simSnap =
  mxCreateStructMatrix(1, 1, nFields, fieldNames);
return simSnap;
}
```

Languages C, C++, M

See Also `mdlSetSimState`

Purpose

Specify time of the next sample time hit

Required

No

C Syntax

```
#define MDL_GET_TIME_OF_NEXT_VAR_HIT
void mdlGetTimeOfNextVarHit(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

Description

The Simulink engine 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.

For Level-2 M-file S-functions, use a sample time of -2 to specify a variable sample time. The S-function's output method should then update the `NextTimeHit` property of the instance of the `Simulink.MSFcnRunTimeBlock` class representing the S-Function block to set the time of the next sample time hit. See `/msfcn_vs.m` for an example.

For Level-1 M-file S-functions, a `flag` of `4` is passed to the S-function when the next sample time hit needs to be calculated.

Note The time of the next hit can be a function of the input signals.

C Example

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

Languages C, C++

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

Purpose

Initialize the state vectors of this S-function

Required

No

C Syntax

```
#define MDL_INITIALIZE_CONDITIONS
void mdlInitializeConditions(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
InitializeConditions(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

The Simulink engine 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. In a C MEX S-function, use `ssGetContStates` and/or `ssGetDiscStates` to access the states. In a Level-2 M-file S-function, use the `ContStates` or `Dwork` run-time object methods to access the continuous and discrete states. This method can also perform any other initialization activities that this S-function requires.

Note If you need to ensure that the initialization code in the `mdlInitializeConditions` function is run only once, then move this initialization code into the `mdlStart` method. The MathWorks recommends this code change as a best practice.

If this S-function resides in an enabled subsystem configured to reset states, the Simulink engine also calls this method when the enabled subsystem restarts execution. C MEX S-functions can use the `ssIsFirstInitCond` macro to determine whether the time at which mdlInitializeCondition is called is equal to the simulation start time.

Note When generating code for a noninlined C MEX S-function that contains this method, make sure the method is not wrapped in a `#if defined(MATLAB_MEX_FILE)` statement. For example:

```
#define MDL_INITIALIZE_CONDITIONS
#if defined(MDL_INITIALIZE_CONDITIONS) && defined(MATLAB_MEX_FILE)
static void mdlInitializeConditions(SimStruct *S)
{
   /* Add mdlInitializeConditions code here *
}
#endif
```

The `define` statement makes the `mdlInitializeConditions` method available only to a MATLAB MEX-file. If the S-function is not inlined, the Real-Time Workshop product cannot use this method, resulting in link or run-time errors.

The Simulink engine calls `mdlInitializeConditions` prior to calculating the S-function's input signals. Therefore, since the input signal values are not yet available, `mdlInitializeConditions` should not use the input signal values to set initial conditions. If your S-function needs to initialize internal values using the block's input signals, perform the initialization in `mdlOutputs`.

For example, in a C MEX S-function, initializes an IWork vector with one element in the `mdlInitializeSizes` method.

```
ssSetNumIWork(S, 1);
```

The IWork vector holds a flag indicating if initial values have been specified. Initialize the flag's value in the `mdlInitializeCondition` method.

```
static void mdlInitializeConditions(SimStruct *S)
{
  /* The mdlInitializeConditions method is called when the simulation
     start and every time an enabled subsystem is re-enabled.

     Reset the IWork flag to 1 when values need to be reinitialized.*/

  ssSetIWorkValue(S, 0, 1);
}
```

Check the value of the IWork vector flag in the `mdlOutputs` method, to determine if initial values need to be set. Since the engine has calculated input values at this point in the simulation, the `mdlOutputs` method can use them to initialize internal values.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    // Initialize values if the IWork vector flag is true. //
    if (ssGetIWorkValue(S, 0) == 1) {
             // Enter initialization code here //
    }

    // Remainder of mdlOutputs function //
}
```

For a Level-2 M-file S-function, use a DWork vector instead of an IWork vector in the previous example.

C Example

This example initializes both a continuous and discrete state 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 `resetint.c`.

M Example

This example initializes both a continuous and discrete state to 1.0. Level-2 M-file S-functions store discrete states in their DWork vectors.

```
function InitializeConditions(s)

s.ContStates.Data(1) = 1;
s.Dwork(1).Data      = 1;

% endfunction
```

Languages

C, C++, M

See Also

`mdlStart`, `mdlOutputs`, `ssIsFirstInitCond`, `ssGetContStates`, `ssGetDiscStates`, `ssGetTStart`, `ssGetT`

Purpose

Specify the sample rates at which this S-function operates

Required

Yes

C Syntax

```
#define MDL_INITIALIZE_SAMPLE_TIMES
void mdlInitializeSampleTimes(SimStruct *S)
```

C 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_types.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, the Simulink engine 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]
```

The Simulink engine 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, the engine 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) {
}
```

Note When generating code for a noninlined S-function that contains this method, make sure the method is not wrapped in a `#if defined(MATLAB_MEX_FILE)` statement. For example:

```
#if defined(MATLAB_MEX_FILE)
static void mdlInitializeSampleTimes(SimStruct *S)
{
   /* Add mdlInitializeSampleTimes code here *
}
#endif
```

The `define` statement makes the `mdlInitializeSampleTimes` method available only to a MATLAB MEX-file. If the S-function is not inlined, the Real-Time Workshop product cannot use this method, resulting in link or run-time errors.

Languages

C, C++

See Also

`mdlSetInputPortSampleTime`, `mdlSetOutputPortSampleTime`

mdlInitializeSizes

Purpose

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

Required

Yes

C Syntax

```
#define MDL_INITIAL_SIZES
void mdlInitializeSizes(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
setup(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

C Description

This is the first S-function callback methods that the Simulink engine calls. This method performs 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, the engine issues an error during simulation (or when in external mode when using the Real-Time Workshop product) 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, including:

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

    See `ssSetInputPortDimensionInfo` 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 the Simulink engine that `u` is not used in either of these S-function routines. Violating this leads to unpredictable results.

- Configure the block's output ports, including:
  - 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 (for example, the outputs only take specific values such as 0, 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 8-33 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 Chapter 12, "S-Function Options — Alphabetical List" for information on each option. Use a bitwise `OR` operator to set multiple options, as in

  ```
  ssSetOptions(S, (SS_OPTION_name1 | SS_OPTION_name2))
  ```

Note When generating code for a noninlined S-function that contains this method, make sure the method is not wrapped in a `#if defined(MATLAB_MEX_FILE)` statement. For example:

```
#if defined(MATLAB_MEX_FILE)
static void mdlInitializeSizes(SimStruct *S)
{
   /* Add mdlInitializeSizes code here *
}
#endif
```

The `define` statement makes the `mdlInitializeSizes` method available only to a MATLAB MEX-file. If the S-function is not inlined, the Real-Time Workshop product cannot use this method, resulting in link or run-time errors.

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 the Simulink engine 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`.

M Description

The Level-2 M-file S-function `setup` method performs nearly the same tasks as the C MEX S-function `mdlInitializeSizes` method, with two significant differences. The `setup` method does not initialize discrete state information, but it does specify the block sample times, eliminating the need for an `mdlInitializeSampleTimes` method. Use the following properties and methods of the run-time object `s` to configure the S-function:

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

  Use `s.DialogPrmsTunable` to set the tunablility of each dialog parameter. When a parameter has been specified as not tunable, the Simulink engine issues an error during simulation (or when in external mode when using the Real-Time Workshop product) if an attempt is made to change the parameter.
- Specify the number of continuous states that this function has, using `s.NumContStates`. Specify discrete state information in the `PostPropagationSetup` method using a DWork vector.
- Configure the block's input ports, including:

- Specify the number of input ports that this S-function has, using `s.NumInputPorts`.
- Specify the dimensions of the *`i`*`th` input port, using `s.InputPort(`*`i`*`).Dimensions`.
- If using port-based sample times, specify the sample time of the *`i`*`th` input port, using `s.InputPort(`*`i`*`).SampleTime`.
- For each input port, specify whether it has direct feedthrough, using `s.InputPort(`*`i`*`).DirectFeedthrough`.

  A port has direct feedthrough if the input is used in the `Outputs` method to calculate the output or the next sample time, for an S-function with a variable sample time. 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 `Outputs` method. Setting the direct feedthrough flag to 0 tells the engine that `u` is not used in this S-function method. Violating this leads to unpredictable results.

See `Simulink.BlockData` and its parent and children classes for a list of all the properties and methods associated with a Level-2 M-file S-function input port.

- Configure the block's output ports, including:
  - Specify the number of output ports that the block has, using `s.NumOutputPorts`.
  - Specify the dimensions of the *`i`*`th` output port, using `s.OutputPort(`*`i`*`).Dimensions`.
  - If using port-based sample times, specify the sample time of the *`i`*`th` output port, using `s.OutputPort(`*`i`*`).SampleTime`.
- Set the block-based sample times (i.e., sample rates), using `s.SampleTimes`.

  See "Sample Times" on page 8-33 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.

See "Using the `setup` Method" on page 3-8 for additional information and examples using the `setup` method.

C 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 feedthrough     */

    int_T inputPortIdx  = 0;
    int_T outputPortIdx = 0;

    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /*
         * If the number of expected input parameters is not
         * equal to the number of parameters entered in the
         * dialog box, return. The Simulink engine generates 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 */
```

Languages C, C++, M

Purpose

Compute the signals that this block emits

Required

Yes

C Syntax

```
#define MDL_OUTPUTS
void mdlOutputs(SimStruct *S, int_T tid)
```

C Arguments

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

`tid`
: Task ID.

M Syntax

```
Outputs(s)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-file S-Function block.

C Description

The Simulink engine 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 8-45).

Use the `UNUSED_ARG` macro if the S-function does not contain task-specific blocks of code to indicate that the `tid` input argument is required but not used in the body of the callback. To do this, insert the line

```
UNUSED_ARG(tid)
```

after the declarations in `mdlOutputs`.

Note When generating code for a noninlined S-function that contains this method, make sure the method is not wrapped in a `#if defined(MATLAB_MEX_FILE)` statement. For example:

```
#if defined(MATLAB_MEX_FILE)
static void mdlOutputs(SimStruct *S)
{
   /* Add mdlOutputs code here *
}
#endif
```

The `define` statement makes the `mdlOutputs` method available only to a MATLAB MEX-file. If the S-function is not inlined, the Real-Time Workshop product cannot use this method, resulting in link or run-time errors.

M Description

In a Level-2 M-file S-function, the `Outputs` method calculates the S-function's outputs at the current time step and store the results in the run-time object's `OutputPort(`*`n`*`).Data` property. In addition, for S-functions with a variable sample time, the `Outputs` method computes the next sample time hit.

Use the run-time object method `IsSampleHit` to determine if the current simulation time is one at which a task handled by this block is active. For port-based sample times, use the `IsSampleHit` property of the run-time object's `InputPort` or `OutputPort` methods to determine if the port produces outputs or accepts inputs at the current simulation time step.

Set the run-time object's `NextTimeHit` property to specify the time of the next sample hit for variable sample-time S-functions.

C Example

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

Languages

C, C++, M

See Also

ssGetOutputPortComplexSignal, ssGetOutputPortRealSignal, ssGetOutputPortSignal

Purpose

Process the S-function's parameters

Required

No

C Syntax

```
#define MDL_PROCESS_PARAMETERS
void mdlProcessParameters(SimStruct *S)
```

C Arguments

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

M Syntax

```
ProcessParameters(s)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

This is an optional routine that the Simulink engine 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 function is only valid for simulation. C MEX S-functions must enclose the method in a `#if defined(MATLAB_MEX_FILE)` statement.

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

C 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 */
```

Languages C, C++, M

See Also `mdlCheckParameters`

Purpose

Perturb the solver's solution of a system's states to better satisfy time-invariant solution relationships

Required

No

C Syntax

```
#define MDL_PROJECTION
void mdlProjection(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
Projection(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

This method is intended for use with S-functions that model dynamic systems whose states satisfy time-invariant relationships, such as those resulting from mass or energy conservation or other physical laws. The Simulink engine invokes this method at each time step after the model's solver has computed the S-function's states for that time step. Typically, slight errors in the numerical solution of the states cause the solutions to fail to satisfy solution invariants exactly. Your `mdlProjection` method can compensate for the errors by perturbing the states so that they more closely approximate solution invariants at the current time step. As a result, the numerical solution adheres more closely to the ideal solution as the simulation progresses, producing a more accurate overall simulation of the system modeled by your S-function.

Your `mdlProjection` method's perturbations of system states must fall within the solution error tolerances specified by the model in which the S-function is embedded. Otherwise, the perturbations may invalidate the solver's solution. It is up to your `mdlProjection` method to ensure that the perturbations meet the error tolerances specified by the model.

See “Perturbing a System’s States Using a Solution Invariant” on page 9-39 for a simple method for perturbing a system’s states. The following articles describe more sophisticated perturbation methods that your `mdlProjection` method can use.

- C.W. Gear, “Maintaining Solution Invariants in the Numerical Solution of ODEs,” *Journal on Scientific and Statistical Computing*, Vol. 7, No. 3, July 1986.
- L.F. Shampine, “Conservation Laws and the Numerical Solution of ODEs I,” *Computers and Mathematics with Applications*, Vol. 12B, pp. 1287–1296, 1986.
- L.F. Shampine, “Conservation Laws and the Numerical Solution of ODEs II,” *Computers and Mathematics with Applications*, Vol. 38, pp. 61–72, 1999.

Perturbing a System’s States Using a Solution Invariant

Here is a simple, Taylor-series-based approach to perturbing a system’s states. Suppose your S-function models a dynamic system having a solution invariant, $g(X,t)$, i.e., g is a continuous, differentiable function of the system states, X, and time, t, whose value is constant with time. Then

$$X_n \cong X_n^* + J_n^T (J_n J_n^T)^{-1} R_n$$

where

- X_n is the system’s ideal state vector at the solver’s current time step
- X_n^* is the approximate state vector computed by the solver at the current time step
- J_n is the Jacobian of the invariant function evaluated at the point in state space specified by the approximate state vector at the current time step:

$$J_n = \frac{\partial g}{\partial X}(X_n^*, t_n)$$

- t_n is the time at the current time step
- R_n is the residual (difference) between the invariant function evaluated at X_n and X_n^* at the current time step:

$$R_n = g(X_n, t_n) - g(X_n^*, t_n)$$

Note The value of $g(X_n, t_n)$ is the same at each time step and is known by definition.

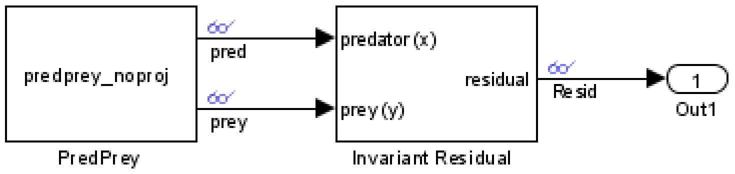
Given a continuous, differentiable invariant function for the system that your S-function models, this formula allows your S-function's `mdlProjection` method to compute a perturbation

$$J_n^T(J_n J_n^T)^{-1} R_n$$

of the solver's numerical solution, X_n^*, that more closely matches the ideal solution, X_n, keeping the S-function's solution from drifting from the ideal solution as the simulation progresses.

M Example

This example illustrates how the perturbation method outlined in the previous section can keep a model's numerical solution from drifting from the ideal solution as a simulation progresses. Consider the following model (open):



The PredPrey block references an S-function, `predprey_noproj.m`, that uses the Lotka-Volterra equations

$$\dot{x} = ax(1-y)$$
$$\dot{y} = -cy(1-x)$$

to model predator-prey population dynamics, where $x(t)$ is the population density of the predators and $y(t)$ is the population density of prey. The ideal solution to the predator-prey ODEs satisfies the time-invariant function

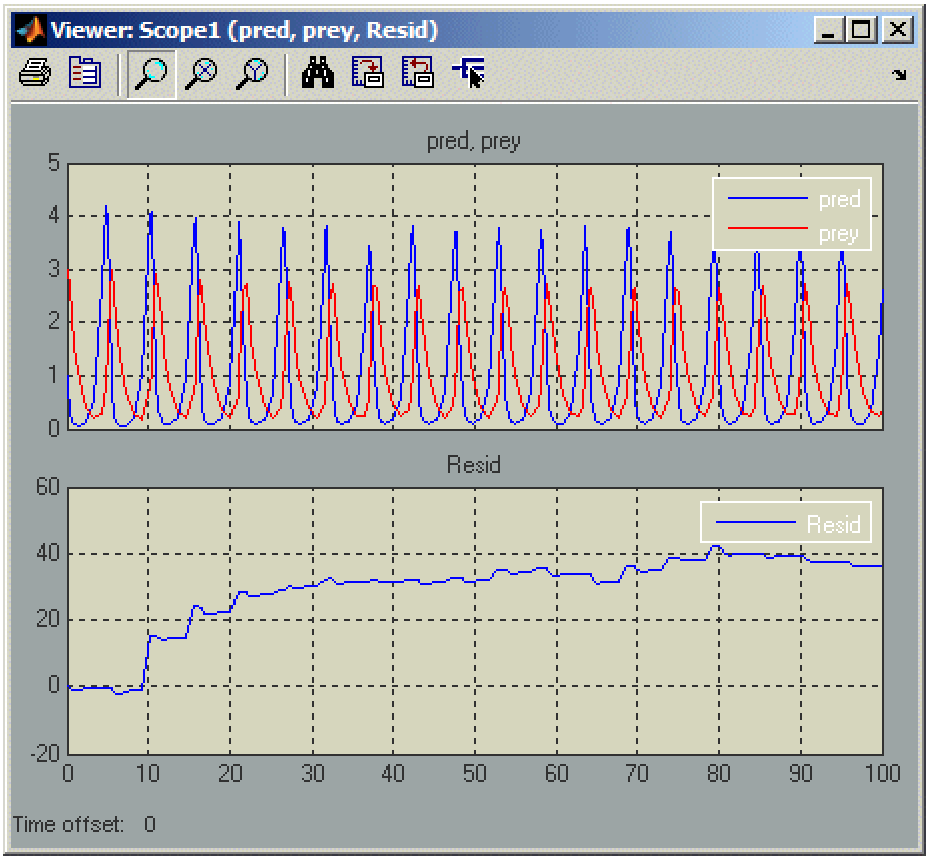
$$x^{-c}e^{cx}y^{-a}e^{ay} = d$$

where a, c, and d are constants. The S-function assumes `a = 1`, `c = 2`, and `d = 121.85`.

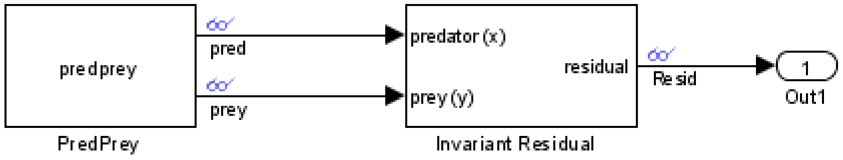
The Invariant Residual block in this model computes the residual between the invariant function evaluated along the system's ideal trajectory through state space and its simulated trajectory:

$$R_n = d - x_n^{-c}e^{cx_n}y_n^{-a}e^{ay_n}$$

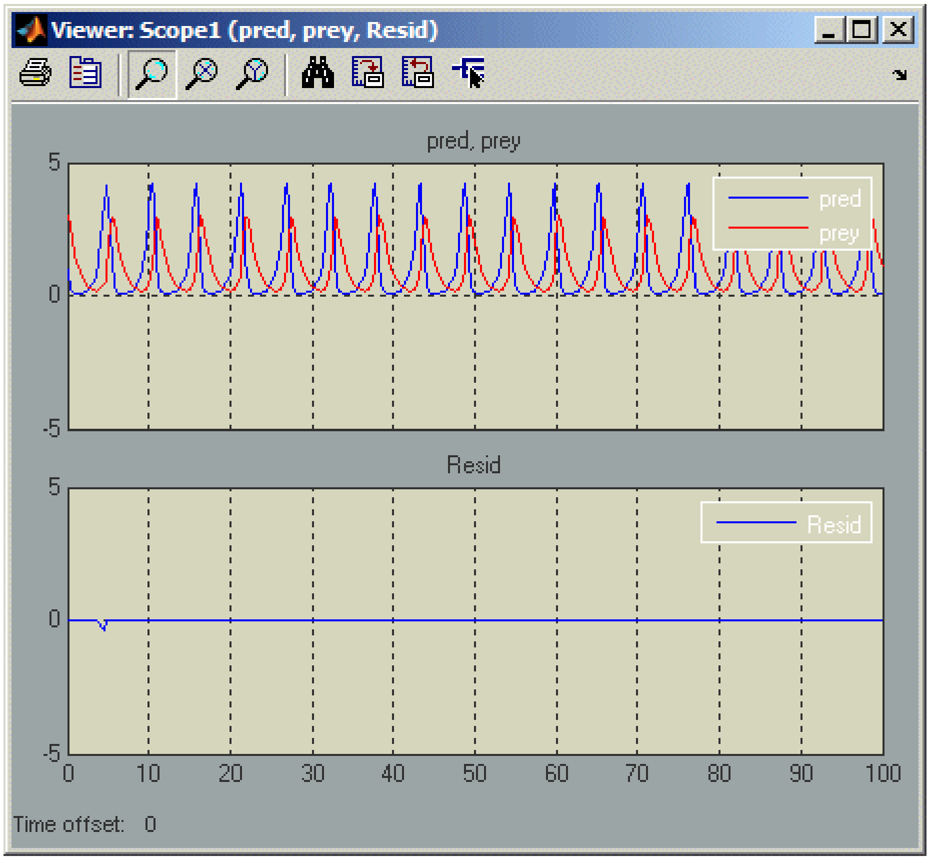
where x_n and y_n are the values computed by the model's solver for the predator and prey population densities, respectively, at the current time step. Ideally, the residual should be zero throughout simulation of the model, but simulating the model reveals that the residual actually strays considerably from zero:



Now consider the following model (open):



This model is the same as the previous model, except that its S-function, `predprey.m`, includes a `mdlProjection` method that uses the perturbation approach outlined in “Perturbing a System’s States Using a Solution Invariant” on page 9-39 to compensate for numerical drift. As a result, the numerical solution more closely tracks the ideal solution as the simulation progresses as demonstrated by the residual signal, which remains near or at zero throughout the simulation:



Languages

C, C++, M

mdlRTW

Purpose

Generate code generation data

Required

No

C Syntax

```
#define MDL_RTW
void mdlRTW(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
WriteRTW(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

This function is called when the Real-Time Workshop product is generating the *`model`*`.rtw` file. In C MEX S-functions, 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`

In C MEX S-functions, this function must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

In Level-2 M-file S-functions, use the run-time object's `WriteRTWParam` method to write custom parameters to the `*model*.rtw` file.

Languages

C, C++, M

C Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multiport.c` in the Simulink model `sldemo_msfcn_lms.mdl` for an example.

M Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sfcndemo_sfun_multiport.mdl` for an example.

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

Required

No

C Syntax

```
#define MDL_SET_DEFAULT_PORT_COMPLEX_SIGNALS
void mdlSetDefaultPortComplexSignals(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

Description

The Simulink engine 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. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

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

Languages

C, C++

See Also

`ssSetOutputPortComplexSignal`, `ssSetInputPortComplexSignal`

Purpose

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

Required

No

C Syntax

```
#define MDL_SET_DEFAULT_PORT_DATA_TYPES
void mdlSetDefaultPortDataTypes(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

Description

The Simulink engine invokes this method if the block has ports whose data types cannot be determined from block connectivity. (This usually happens when the block is unconnected or is part of a feedback loop.) This method must set the data types of all ports whose data types are not set. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

If the block does not implement this method and the engine cannot determine the data types of any of its ports, the engine sets the data types of all the ports to `double`. If the block does not implement this method and the engine cannot determine the data types of some, but not all, of its ports, the engine sets the unknown ports to the data type of the port whose data type has the largest size.

The engine invokes an error if the `mdlSetDefaultPortDataType` method attempts to modify the data type of a port when the data type was previously specified by `mdlSetInputPortDataType` or `mdlSetOutputPortDataType`. If an S-function has multiple input or output ports, `mdlSetDefaultPortDataType` should check if the data type of a port is still dynamic before attempting to set the type. For example, the `mdlSetDefaultPortDataType` uses the following lines to check if the data type of the second input port is still unknown.

```
if (ssGetInputPortDataType(S, 1) == DYNAMICALLY_TYPED) {
   ssSetInputPortDataType(S, 1, SS_UINT8 );
```

```
    }
```

Languages C, C++

See Also `ssSetOutputPortDataType`, `ssSetInputPortDataType`

Purpose

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

Required

No

C Syntax

```
#define MDL_SET_DEFAULT_PORT_DIMENSION_INFO
void mdlSetDefaultPortDimensionInfo(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

Description

The Simulink engine 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. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

If the S-function does not implement this method, the engine tries to find a set of dimensions that will satisfy the dimension propagation rules implemented using `mdlSetInputPortDimensionInfo` and `mdlSetOutputPortDimensionInfo`. This process might not be able to produce a valid set of dimensions for S-functions with special dimension requirements.

The engine invokes an error if the `mdlSetDefaultPortDimensionInfo` method attempts to modify the dimensions of a port when the dimensions were previously specified by `mdlSetInputPortDimensionInfo` or `mdlSetOutputPortDimensionInfo`. If an S-function has multiple input or output ports, `mdlSetDefaultPortDimensionInfo` should check if the dimensions of the port are still dynamic before attempting to set the dimensions. For example, the `mdlSetDefaultPortDimensionInfo` uses the following lines to check if the dimensions of the first output port are still unknown.

```
        if (ssGetOutputPortWidth(S, 0) == DYNAMICALLY_SIZED) {
           ssSetOutputPortMatrixDimensions(S, 0, 1, 1 );
        }
```

Example See *matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_matadd.c for an example of how to use this function.

Languages C, C++

See Also ssSetErrorStatus, ssSetOutputPortMatrixDimensions

Purpose

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

Required

No

C Syntax

```
#define MDL_SET_INPUT_PORT_COMPLEX_SIGNAL
void mdlSetInputPortComplexSignal(SimStruct *S, int_T port,
 CSignal_T csig)
```

C Arguments

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

`port`
: Index of a port.

`csig`
: Numeric type of signal, either `COMPLEX_NO` (real) or `COMPLEX_YES` (complex).

M Syntax

```
SetInputPortComplexSignal(s, port, csig)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

`csig`
: Integer value specifying whether the port accepts real (`false` or `0`) or complex (`true` or `1`) signals.

Description

The Simulink engine calls this routine to set the input port numeric type for inputs that have this attribute set to `COMPLEX_INHERITED`. The input `csig` is the proposed numeric type for this input port. This method is only valid for simulation. C MEX S-functions must enclosed this method in a `#if defined(MATLAB_MEX_FILE)` statement.

The S-function must check whether the proposed numeric type is a valid type for the specified port. If it is valid, a C MEX S-function sets the numeric type of the specified input port using `ssSetInputPortComplexSignal`. Otherwise, it reports an error using `ssSetErrorStatus`.

Level-2 M-file S-functions set the numeric type of the specified input port using the line

```
s.InputPort(port).Complexity = csig;
```

The S-function can also set the numeric types of other input and output ports with inherited numeric types. The engine reports an error if the S-function changes the numeric type of a port whose numeric type is known.

If the S-function does not implement this routine, the engine assumes that the S-function accepts a real or complex signal and sets the input port numeric type to the specified value.

The engine calls this method until all input ports with inherited numeric types have their numeric types specified.

C Example

See *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` for an example of how to use this function.

Languages

C, C++, M

See Also

`ssSetErrorStatus`, `ssSetInputPortComplexSignal`

Purpose

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

Required

No

C Syntax

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

C Arguments

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

`port`
: Index of a port.

`id`
: Data type ID.

M Syntax

```
SetInputPortDataType(s, port, id)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

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

Description

The Simulink engine calls this routine to set the data type of `port` when `port` has an inherited data type. The data type `id` is the proposed data type for this port. Data type IDs for the built-in data types can be found in `*matlabroot*/simulink/include/simstruc_types.h`. This method is only valid for simulation. C MEX S-functions must enclose this method in a `#if defined(MATLAB_MEX_FILE)` statement.

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, a C MEX S-functions sets the data type of the input port using `ssSetInputPortDataType`. Otherwise, it reports an error using `ssSetErrorStatus`.

Level-2 M-file S-functions set the data type of the input port using the line

```
s.InputPort(port).DatatypeID = id;
```

The S-function can also set the data types of other input and output ports if they are unknown. The engine 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, the engine assumes that the block accepts any data type and sets the input port data type to the specified value.

The engine calls this method until all input ports with inherited data types have their data types specified.

Languages

C, C++, M

See Also

`ssSetErrorStatus`, `ssSetInputPortDataType`

Purpose

Set the dimensions of the signals accepted by an input port

Required

No

C Syntax

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

C Arguments

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

`port`
: Index of a port.

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

See `ssSetInputPortDimensionInfo` for a description of this structure.

M Syntax

```
SetInputPortDimensions(s, port, dimsInfo)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

`dimsInfo`
: Array that specifies the signal dimensions supported by the port, e.g., [5] for a 5-element vector signal or [3 3] for a 3-by-3 matrix signal.

Description

The Simulink engine calls this method during dimension propagation with candidate dimensions `dimsInfo` for `port`. In C MEX S-functions, if the proposed dimensions are acceptable, the method sets the actual port dimensions, using `ssSetInputPortDimensionInfo`. If they are unacceptable, the method generates an error via `ssSetErrorStatus`.

A Level-2 M-file S-function sets the input port dimensions using the line

```
s.InputPort(port).Dimensions = dimsInfo;
```

This method is only valid for simulation. A C MEX S-function must enclose the method in a `#if defined(MATLAB_MEX_FILE)` statement.

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

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

The engine calls this method until all input ports with inherited dimensions have their dimensions specified.

C Example

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

Languages

C, C++, M

See Also

`ssSetErrorStatus`, `mdlSetOutputPortDimensionInfo`

mdlSetInputPortDimensionsModeFcn

Purpose

Propagate the dimensions mode

Required

No

C Syntax

```
void mdlSetInputPortDimensionsModeFcn(SimStruct *S, int_T portIdx,
DimensionsMode_T dimsMode)
```

C Arguments

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

`portIdx`
: Index of a port.

`dimsMode`
: Current dimensions mode. Possible values are INHERIT_DIMS_MODE FIXED_DIMS_MODE and VARIABLE_DIMS_MODE

M Syntax

```
SetInputPortDimensionsMode(s, port, dm)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

`dm`
: Integer value representing the dimensions mode of the port.

Description

The Simulink engine calls this optional method to enable this S-function to set the dimensions mode of the input port indexed by portIdx.

C Example

See
`*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_hol`
for an example of how to use this function.

Languages C, C++, M

Purpose

Specify whether an input port accepts frame data

Required

No

C Syntax

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

C Arguments

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

`port`
: Index of a port.

`frameData`
: String indicating if the input port accepts frame data.

M Syntax

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

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

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

Description

This method is called with the candidate frame setting for an input port.

For C MEX S-functions, `frameData` is either `FRAME_YES` or `FRAME_NO`. If the proposed setting is acceptable, the method sets the actual frame data setting using `ssSetInputPortFrameData`. If the setting is unacceptable, the method generates an error via `ssSetErrorStatus`.

Note that any other 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 configured. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

For Level-2 M-file S-functions, if the value of `mode` is acceptable, this method sets the sampling mode using the line

```
s.InputPort(port).SamplingMode = mode;
```

The Simulink engine calls this method until all input ports with inherited frame settings have their frame settings specified.

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

Languages

C, C++, M

See Also

`ssSetInputPortFrameData`, `ssSetOutputPortFrameData`, `ssSetErrorStatus`

mdlSetInputPortSampleTime

Purpose

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

Required

No

C Syntax

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

C Arguments

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

`port`
: Index of a port.

`sampleTime`
: Inherited sample time for `port`.

`offsetTime`
: Inherited offset time for `port`.

M Syntax

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

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

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

Description

The Simulink engine invokes this method with the sample time that `port` inherits from the port to which it is connected.

For C MEX S-functions, if the inherited sample time is acceptable, this method sets the sample time of `port` to the inherited time, using `ssSetInputPortSampleTime` and `ssSetInputPortOffsetTime`. If the sample time is unacceptable, this method generates an error via `ssSetErrorStatus`. Note that any other 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`. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

For Level-2 M-file S-functions, if the inherited sample time is acceptable, this method sets the sample time and offset time using the line

```
s.InputPort(port).SampleTime = time;
```

The engine calls this method until all input ports with inherited sample times are specified.

When inherited port-based sample times are specified, the sample time is guaranteed to be one of the following where `0.0 < period < inf` and `0.0 <= offset < period`.

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

Constant, triggered, and variable-step sample times are not propagated to S-functions with port-based sample times.

Generally `mdlSetInputPortSampleTime` is called once per port 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, the Simulink engine converts the sample time to the fundamental sample time for the model.
- The engine adjusts the sample time to be as numerically sound as possible. For example, the engine converts `[0.2499999999999, 0]` to `[0.25, 0]`.

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

Languages

C, C++, M

See Also

`ssSetInputPortSampleTime`, `ssSetOutputPortSampleTime`, `mdlInitializeSampleTimes`

Purpose

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

Required

No

C Syntax

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

C 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 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`. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

The Simulink engine invokes this method until all dynamically sized input ports are configured.

Languages

C, C++

See Also

`ssSetInputPortWidth`, `ssSetOutputPortWidth`, `ssSetErrorStatus`

Purpose

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

Required

No

C Syntax

```
#define MDL_SET_OUTPUT_PORT_COMPLEX_SIGNAL
void mdlSetOutputPortComplexSignal(SimStruct *S, int_T port,
 CSignal_T csig)
```

C Arguments

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

`port`
: Index of a port.

`csig`
: Numeric type of signal, either `COMPLEX_NO` (real) or `COMPLEX_YES` (complex).

M Syntax

```
SetOutputPortComplexSignal(s, port, csig)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

`csig`
: Integer value specifying whether the port produces real (0) or complex (1) signals.

Description

The Simulink engine calls this routine to set the output port numeric type for outputs that have this attribute set to `COMPLEX_INHERITED`. The input argument `csig` is the proposed numeric type for this output port. The S-function must check whether the specified numeric type is a valid type for the specified port.

If it is valid, C MEX S-functions set the numeric type of the specified output port using `ssSetOutputPortComplexSignal`. Otherwise, the S-function reports an error, using `ssSetErrorStatus`. This method is only valid for simulation. C MEX S-functions must enclose the method in a `#if defined(MATLAB_MEX_FILE)` statement.

Level-2 M-file S-functions set the numeric type of the specified output port using the line

```
s.OutputPort(port).Complexity = csig;
```

The S-function can also set the numeric types of other input and output ports with unknown numeric types. The engine reports an error if the S-function changes the numeric type of a port whose numeric type is known.

If the S-function does not implement this routine, the engine assumes that the S-function accepts a real or complex signal and sets the output port numeric type to the specified value.

The engine calls this method until all output ports with inherited numeric types have their numeric types specified.

Example

See *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` for an example of how to use this function.

Languages

C, C++, M

See Also

`ssSetOutputPortComplexSignal`, `ssSetErrorStatus`

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

Required No

C Syntax

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

C Arguments

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

`port`
: Index of an output port.

`id`
: Data type ID.

M Syntax

```
SetOutputPortDataType(s, port, id)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

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

Description The Simulink engine calls this routine to set the data type of `port` when `port` has an inherited data type. The data type ID `id` is the proposed data type for this port. Data type IDs for the built-in data types can be found in `*matlabroot*/simulink/include/simstruc_types.h`. 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, a C MEX S-function sets the data type of `port` using `ssSetOutputPortDataType`. Otherwise, the S-function reports an error, using `ssSetErrorStatus`. This method is only valid for simulation. C MEX S-functions must enclose the method in a `#if defined(MATLAB_MEX_FILE)` statement.

Level-2 M-file S-functions set the data type of the output port using the line

```
s.OutputPort(port).DatatypeID = id;
```

The S-function can also set the data types of other input and output ports if their data types have not been set. The engine 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, the engine assumes that the block supports any data type and sets the output port data type to the specified value.

The engine calls this method until all output ports with inherited data types have their data types specified.

Languages

C, C++, M

See Also

`ssSetOutputPortDataType`, `ssSetErrorStatus`

Purpose

Set the dimensions of the signals accepted by an output port

Required

No

C Syntax

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

C Arguments

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

`port`
: Index of a port.

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

See `ssSetInputPortDimensionInfo` for a description of this structure.

M Syntax

```
SetOutputPortDimensions(s, port, dimsInfo)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

`dimsInfo`
: Array that specifies the signal dimensions supported by the port, e.g., [5] for a 5-element vector signal or [3 3] for a 3-by-3 matrix signal.

Description

The Simulink engine calls this method with candidate dimensions `dimsInfo` for `port`. In C MEX S-functions, if the proposed dimensions are acceptable, the method sets the actual port dimensions, using `ssSetOutputPortDimensionInfo`. If they are unacceptable, the method generates an error via `ssSetErrorStatus`. This method is only valid for simulation. C MEX S-functions must enclose the method in a `#if defined(MATLAB_MEX_FILE)` statement.

A Level-2 M-file S-function sets the output port dimensions using the line

```
  s.OutputPort(port).Dimensions = dimsInfo;
```

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

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

The engine calls this method until all output ports with inherited dimensions have their dimensions specified.

Example

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

Languages

C, C++, M

See Also

`ssSetErrorStatus`, `ssSetOutputPortDimensionInfo`

Purpose

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

Required

No

C Syntax

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

C Arguments

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

`port`
: Index of a port.

`sampleTime`
: Inherited sample time for `port`.

`offsetTime`
: Inherited offset time for `port`.

M Syntax

```
SetOutputPortSampleTime(s, port, time)
```

M Arguments

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

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

`time`
: Two-element array, `[period offset]`, that specifies the period and offset of the times that this port produces output.

Description

The Simulink engine calls this method with the sample time that `port` inherits from the port to which it is connected.

For C MEX S-functions, if the inherited sample time is acceptable, this method should set the sample time of `port` to the inherited sample time and offset time, using `ssSetOutputPortSampleTime` and `ssSetOutputPortOffsetTime`. If the sample time is unacceptable, this method generates an error via `ssSetErrorStatus`. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

For Level-2 M-file S-functions, if the inherited sample time is acceptable, this method sets the sample time and offset time using the line

```
s.OutputPort(port).SampleTime = time;
```

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` in C MEX S-functions, or setting the `SampleTime` property of the `Simulink.BlockPortData` object associated with the port in Level-2 M-file S-functions.

Normally, sample times are propagated forward; however, if sources feeding this block have inherited sample times, the engine might choose to back-propagate known sample times to this block. When back-propagating sample times, this method is called in succession for all inherited output port signals.

See `mdlSetInputPortSampleTime` for more information about when this method is called.

Languages

C, C++, M

See Also

`ssSetOutputPortSampleTime`, `ssSetErrorStatus`, `ssSetInputPortSampleTime`, `ssSetOutputPortSampleTime`, `mdlSetInputPortSampleTime`, `Simulink.MSFcnRunTimeBlock`, `Simulink.BlockPortData`

Purpose

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

Required

No

C Syntax

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

C 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`. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

Languages

C, C++

See Also

`ssSetInputPortWidth`, `ssSetOutputPortWidth`, `ssSetErrorStatus`

Purpose

Set the simulation state of the S-function by restoring the SimState.

Required

No

C Syntax

```
#define MDL_SIM_STATE
void mdlSetSimState(SimStruct* S, const mxArray* in)
```

C Arguments

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

`const mxArray* in`
: Any valid MATLAB data.

M Syntax

```
SetSimState(s, in)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

`in`
: The MATLAB data of type returned by `GetSimState`.

Description

The Simulink engine invokes this custom method at the beginning of a simulation of the model containing `S` . Simulink sets the initial simulation state of the S-function to the SimState of the model.

C Example

```
/* Function: mdlSetSimState
 * Abstract:
 *   Unpack the MATLAB structure passed and restore it to
 *   the RunTimeData structure
 */
static void mdlSetSimState(SimStruct* S,
const mxArray* simSnap)
{
    unsigned n = (unsigned)(ssGetInputPortWidth(S, 0));
 RunTimeData_T* rtd =
(RunTimeData_T*)ssGetPWorkValue(S, 0);

 /* Check and load the count value */
 {
     const mxArray* cnt =
 mxGetField(simSnap, 0, fieldNames[0]);
     ERROR_IF_NULL(S,cnt,
 "Count field not found in simulation state");
     if ( mxIsComplex(cnt) ||
          !mxIsUint64(cnt) ||
          mxGetNumberOfElements(cnt) != 1 ) {
         ssSetErrorStatus(S, "Count field is invalid");
         return;
     }
     rtd->cnt = ((uint64_T*)(mxGetData(cnt)))[0];
 }
```

Languages

C, C++, M

See Also

`mdlInitializeConditions`, `mdlGetSimState`

Purpose

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

Required

No

C Syntax

```
#define MDL_SET_WORK_WIDTHS
void mdlSetWorkWidths(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
PostPropagationSetup(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

The Simulink engine 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 8-8). The engine 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`.

A C-MEX S-function needs to implement this method only if it does not know the sizes of all the work vectors it requires when the engine 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`. This method is only valid for simulation, and must be enclosed in a `#if defined(MATLAB_MEX_FILE)` statement.

A Level-2 M-file S-function must implement this method if any DWork vectors are used in the S-function. In the case of M-file S-functions, this method sets the number of DWork vectors and initializes their attributes. For example, the following code in the `PostPropagationSetup` method specifies the usage for the first DWork vector:

```
s.DWork(1).Usage = type;
```

where `s` is an instance of the `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-file S-Function block and *`type`* is one of the following:

- DWork
- DState
- Scratch
- Mode

C Example

For a full example of a C MEX S-function using DWork vectors, see the file `sfun_rtwdwork.c` used in the Simulink model `sfcndemo_sfun_rtwdwork.mdl`.

M Example

For a full example of a Level-2 M-file S-function using DWork vectors, see the file *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` used in the Simulink model `sldemo_msfcn_lms.mdl`.

Languages

C, C++, M

See Also

`mdlInitializeSizes`

mdlSimStatusChange

Purpose

Respond to a pause or resumption of the simulation of the model that contains this S-function

Required

No

C Syntax

```
#define MDL_SIM_STATUS_CHANGE
void mdlSimStatusChange(SimStruct *S,
 ssSimStatusChangeType simStatus)
```

C Arguments

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

`simStatus`
: Status of the simulation, either `SIM_PAUSE` or `SIM_CONTINUE`.

M Syntax

```
SimStatusChange(s, status)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

`status`
: Status of the simulation, either `0` when paused or `1` when continued.

Description

The Simulink engine calls this routine when a simulation of the model containing `S` pauses or resumes. This method is only valid for simulation. C MEX S-functions must enclose the method in a `#if defined(MATLAB_MEX_FILE)` statement.

C Example

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SIM_STATUS_CHANGE
static void mdlSimStatusChange(SimStruct *S,
       ssSimStatusChangeType simStatus) {
  if (simStatus == SIM_PAUSE) {
    ssPrintf("Pause has been called! \n");
  } else if (simStatus == SIM_CONTINUE) {
    ssPrintf("Continue has been called! \n");
  }
}
#endif
```

Languages C, C++

Purpose

Initialize the state vectors of this S-function

Required

No

C Syntax

```
#define MDL_START
void mdlStart(SimStruct *S)
```

C Arguments

`S`

SimStruct representing an S-Function block.

M Syntax

```
Start(s)
```

M Arguments

`s`

Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

The Simulink engine invokes this optional method at the beginning of a simulation. The method performs initialization activities that this S-function requires only once, such as allocating memory, setting up user data, or initializing states.

If your S-function resides in an enabled subsystem and needs to reinitialize its states every whenever the subsystem is enabled, use `mdlInitializeConditions` to initialize the state values, instead of `mdlStart`.

In C MEX S-functions, use `ssGetContStates` and/or `ssGetDiscStates` to get the states.

Languages

C, C++, M

C Example

See *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_directlook.c` for an example of how to use this function.

M Example

See
`matlabroot``/toolbox/simulink/simdemos/simfeatures/msfcn_varpulse.m`
for an example of how to use this function.

See Also

`mdlInitializeConditions`, `ssGetContStates`, `ssGetDiscStates`

mdlTerminate

Purpose Perform any actions required at termination of the simulation

Required Yes

C Syntax

```
void mdlTerminate(SimStruct *S)
```

C Arguments

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

M Syntax

```
Terminate(s)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

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

In C MEX S-functions, the `mdlTerminate` method is called after either an **Update Diagram** or after a simulation for which *one* of the following conditions is met:

- `mdlStart` has run and the S-function does not have any `mdlInitializeConditions` method defined.
- `mdlInitializeConditions` has run.

In addition, if the `SS_OPTION_CALL_TERMINATE_ON_EXIT` option is set for a given S-function, and if `mdlInitializeSizes` is called, then the user is guaranteed that Simulink will call `mdlTerminate`. One reason to set the `SS_OPTION_CALL_TERMINATE_ON_EXIT` option is to allocate memory in `mdlInitializeSizes` rather than wait until `mdlStart`.

Note that Simulink calls `mdlInitializeSizes` under a number of circumstances, including compilation and simulation. Simulink will also call `mdlInitializeSizes` during model editing if you perform an operation such as the setting of parameters.

In C MEX S-functions, use the `UNUSED_ARG` macro if the `mdlTerminate` function does not perform any actions that require the SimStruct `S` to indicate that the `S` input argument is required, but not used in the body of the callback. To do this, insert the line

```
UNUSED_ARG(S)
```

after any declarations in `mdTerminate`.

Note When generating code for a noninlined C MEX S-function that contains this method, make sure the method is not wrapped in a `#if defined(MATLAB_MEX_FILE)` statement. For example:

```
#if defined(MATLAB_MEX_FILE)
static void mdlTerminate(SimStruct *S)
{
   /* Add mdlTerminate code here *
}
#endif
```

The `define` statement makes the `mdlTerminate` method available only to a MATLAB MEX-file. If the S-function is not inlined, the Real-Time Workshop product cannot use this method, resulting in link or run-time errors.

C 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));
      }
    }
  }
```

Languages C, C++, M

See Also `ssSetOptions`

Purpose

Update a block's states

Required

No

C Syntax

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

C Arguments

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

`tid`
: Task ID.

M Syntax

```
Update(s)
```

M Arguments

`s`
: Instance of `Simulink.MSFcnRunTimeBlock` class representing the Level-2 M-File S-Function block.

Description

The Simulink engine 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, the engine is able to eliminate the need for the extra call in these circumstances.

If your C MEX 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.

In C MEX S-functions, 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 8-45).

Use the `UNUSED_ARG` macro if your C MEX S-function does not contain task-specific blocks of code to indicate that the `tid` input argument is required but not used in the body of the callback. To do this, insert the line

```
UNUSED_ARG(tid)
```

after the declarations in `mdlUpdate`.

In Level-2 M-file S-functions, use the run-time object method `IsSampleHit` to determine if the current simulation time is one at which a task handled by this block is active. For port-based sample times, use the `IsSampleHit` property of the run-time object’s `InputPort` or `OutputPort` to determine if the port produces outputs or accepts inputs at the current simulation time step.

Note When generating code for a noninlined C MEX S-function that contains this method, make sure the method is not wrapped in a `#if defined(MATLAB_MEX_FILE)` statement. For example:

```
#define MDL_UPDATE
#if defined(MDL_UPDATE) && defined(MATLAB_MEX_FILE)
static void mdlUpdate(SimStruct *S)
{
   /* Add mdlUpdate code here *
}
#endif
```

The `define` statement makes the `mdlUpdate` method available only to a MATLAB MEX-file. If the S-function is not inlined, the Real-Time Workshop product cannot use this method, resulting in link or run-time errors.

C Example

For an example that uses this function to update discrete states, see *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/dsfunc.c`. For an example that uses this function to update the transfer function coefficients of a time-varying continuous transfer function, see *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c`.

M Example

For an example that uses this function to update discrete states, see *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/msfcn_unit_delay.m`.

Languages

C, C++, M

See Also

`mdlDerivatives`, `ssGetContStates`, `ssGetDiscStates`

Purpose

Update zero-crossing vector

Required

No

C Syntax

```
#define MDL_ZERO_CROSSINGS
void mdlZeroCrossings(SimStruct *S)
```

C 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. Implementing zero-crossing detection typically requires using the zero-crossing and mode work vectors to determine when a zero crossing occurs and how the S-function's outputs should respond to this event. The `mdlZeroCrossings` 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 the Simulink engine 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 the engine with zero-crossing signals, `mdlZeroCrossings` updates the `ssGetNonsampleZCs(S)` vector.

Example

For an example, see *`matlabroot`*`/simulink/src/sfun_zc_sat.c`. A detailed description of this example can be found in "Zero Crossings" on page 8-51 in the "Writing S-Functions" documentation.

Languages C, C++

See Also `mdlInitializeSizes`, `ssGetNonsampledZCs`

SimStruct Functions Reference

Introduction

In this section...

About SimStruct Functions

The Simulink software 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.

Language Support

Some `SimStruct` functions are available only in some of the languages supported by the Simulink software. The reference page for each `SimStruct` macro or function lists the languages in which it is available and gives the syntax for these languages.

Note Most `SimStruct` functions available in C are implemented as C macros. Individual reference pages indicate any `SimStruct` macro that becomes a function when you compile your S-function in debug mode (`mex -g`).

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

In this section...

Data Type

Macro	Description
`ssGetDataTypeId`	Get the ID for a data type.
`ssGetDataTypeIdAliasedThruTo`	Get the ID for the built-in data type associated with a data type alias.
`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.

Macro	Description
ssGetOutputPortDataType	Get the data type of an output port.
ssGetOutputPortSignal	Get an output signal of any type except double.
ssRegisterDataType	Register a data type.
ssSetDataTypeSize	Specify the size of a data type.
ssSetDataTypeZero	Specify the zero representation of a data type.
ssSetInputPortDataType	Specify the data type of signals accepted by an input port.
ssSetOutputPortDataType	Specify the data type of an output port.

Dialog Box Parameters

Macro	Description
ssGetDTypeIdFromMxArray	Get the Simulink data type of a dialog parameter.
ssGetNumSFcnParams	Get the number of parameters that an S-function expects.
ssGetSFcnParam	Get a parameter entered by a user in the S-Function block dialog box.
ssGetSFcnParamsCount	Get the actual number of parameters specified by the user.
ssSetNumSFcnParams	Set the number of parameters that an S-function expects.
ssSetSFcnParamTunable	Specify the tunability of a dialog box parameter.

Error Handling and Status

Macro	Description
`ssGetErrorStatus`	Get a string that identifies the last error.
`ssPrintf`	Print a variable-content `msg`.
`ssSetErrorStatus`	Report errors.
`ssWarning`	Display a warning message.

Function Call

Macro	Description
`ssCallSystemWithTid`	Execute a function-call subsystem connected to an S-function.
`ssDisableSystemWithTid`	Disable a function-call subsystem connected to this S-function block.
`ssEnableSystemWithTid`	Enable a function-call subsystem connected to this S-function.
`ssGetExplicitFCSSCtrl`	Determine whether this S-function explicitly enables and disables the function-call subsystem that it invokes.
`ssSetCallSystemOutput`	Specify that an output port element issues a function call.
`ssSetExplicitFCSSCtrl`	Specify whether an S-function explicitly enables and disables the function-call subsystem that it calls.

Input and Output Ports

I/O Port — Signal Specification

Macro	Description
`ssAllowSignalsWithMoreThan2D`	Enable S-function to work with multidimensional input and output signals.
`ssGetInputPortComplexSignal`	Get the numeric type (complex or real) of an input port.
`ssGetInputPortDataType`	Get the data type of an input port.
`ssGetInputPortDirectFeedThrough`	Determine whether an input port has direct feedthrough.
`ssGetInputPortFrameData`	Determine whether a port accepts signal frames.
`ssGetInputPortOffsetTime`	Determine the offset time of an input port.
`ssGetInputPortRequiredContiguous`	Determine whether the signal elements entering a port must be contiguous.
`ssGetInputPortSampleTime`	Determine the sample time of an input port.
`ssGetInputPortSampleTimeIndex`	Get the sample time index of an input port.
`ssGetOutputPortComplexSignal`	Get the numeric type (complex or real) of an output port.
`ssGetOutputPortDataType`	Get the data type of an output port.
`ssGetOutputPortFrameData`	Determine whether a port outputs signal frames.
`ssGetOutputPortOffsetTime`	Determine the offset time of an output port.

I/O Port — Signal Specification (Continued)

Macro	Description
`ssGetOutputPortSampleTime`	Determine the sample time 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.
`ssSetInputPortDirectFeedThrough`	Specify that an input port is a direct-feedthrough port.
`ssSetInputPortFrameData`	Specify whether a port accepts signal frames.
`ssSetInputPortOffsetTime`	Specify the sample time offset for an input port.
`ssSetInputPortRequiredContiguous`	Specify that the signal elements entering a port must be contiguous.
`ssSetInputPortSampleTime`	Set the sample time of an input port.
`ssSetNumInputPorts`	Set the number of input ports on an S-Function block.
`ssSetNumOutputPorts`	Specify the number of output ports on an S-Function block.
`ssSetOneBasedIndexInputPort`	Specify that an input port expects one-based indices.
`ssSetOneBasedIndexOutputPort`	Specify that an output port emits one-based indices.
`ssSetOutputPortComplexSignal`	Specify the numeric type (real or complex) of this port.
`ssSetOutputPortDataType`	Specify the data type of an output port.

I/O Port — Signal Specification (Continued)

Macro	Description
`ssSetOutputPortFrameData`	Specify whether a port outputs framed data.
`ssSetOutputPortOffsetTime`	Specify the sample time offset value of an output port.
`ssSetOutputPortSampleTime`	Specify the sample time of an output port.
`ssSetZeroBasedIndexInputPort`	Specify that an input port expects zero-based indices.
`ssSetZeroBasedIndexOutputPort`	Specify that an output port emits zero-based indices.

I/O Port — Signal Dimensions

Macro	Description
`ssAllowSignalsWithMoreThan2D`	Enable S-function to work with multidimensional signals.
`ssGetInputPortDimensions`	Get the dimensions of the signal accepted by an input port.
`ssGetInputPortDimensionSize`	Get the size of one dimension of the signal entering an input port.
`ssGetInputPortNumDimensions`	Get the dimensionality of the signals accepted by an input port.
`ssGetInputPortWidth`	Determine the width of an input port.
`ssGetOutputPortDimensions`	Get the dimensions of the signal leaving an output port.

I/O Port — Signal Dimensions (Continued)

Macro	Description
`ssGetOutputPortDimensionSize`	Get the size of one dimension of the signal leaving an output port.
`ssGetOutputPortNumDimensions`	Get the number of dimensions of an output port.
`ssGetOutputPortWidth`	Determine the width of an output port.
`ssSetInputPortDimensionInfo`	Set the dimensionality of an input port.
`ssSetInputPortMatrixDimensions`	Specify dimension information for an input port that accepts matrix signals.
`ssSetInputPortVectorDimension`	Specify dimension information for an input port that accepts vector signals.
`ssSetInputPortWidth`	Set the width of a 1-D (vector) input port.
`ssSetOutputPortDimensionInfo`	Specify the dimensionality of an output port.
`ssSetOutputPortMatrixDimensions`	Specify dimension information for an output port that emits matrix signals.
`ssSetOutputPortVectorDimension`	Specify dimension information for an output port that emits vector signals.
`ssSetOutputPortWidth`	Specify the width of a 1-D (vector) output port.

I/O Port — Signal Dimensions (Continued)

Macro	Description
`ssSetOutputPortMatrixDimensions`	Specify the dimensions of a 2-D (matrix) signal.
`ssSetVectorMode`	Specify the vector mode that an S-function supports.

I/O Port — Signal Access

Macro	Description
`ssGetInputPortBufferDstPort`	Determine the output port that is overwriting an input port's memory buffer.
`ssGetInputPortConnected`	Determine whether an S-Function block port is connected to a nonvirtual block.
`ssGetInputPortOptimOpts`	Determine the reusability setting of the memory allocated to the input port of an S-function.
`ssGetInputPortOverWritable`	Determine whether an input port can be overwritten.
`ssGetInputPortRealSignal`	Get the address of a real, contiguous signal entering an input port.
`ssGetInputPortRealSignalPtrs`	Access the signal elements connected to an input port.
`ssGetInputPortSignal`	Get the address of a contiguous signal entering an input port.
`ssGetInputPortSignalPtrs`	Get pointers to input signal elements of type other than `double`.

I/O Port — Signal Access (Continued)

Macro	Description
`ssGetNumInputPorts`	Can be used in any routine (except `mdlInitializeSizes`) to determine how many input ports a block has.
`ssGetNumOutputPorts`	Can be used in any routine (except `mdlInitializeSizes`) to determine how many output ports a block has.
`ssGetOutputPortConnected`	Determine whether an output port is connected to a nonvirtual block.
`ssGetOutputPortBeingMerged`	Determine whether the output of this block is connected to a Merge block.
`ssGetOutputPortOptimOpts`	Determine the reusability of the memory allocated to the output port of an S-function.
`ssGetOutputPortRealSignal`	Access the elements of a signal connected to an output port.
`ssGetOutputPortSignal`	Get the vector of signal elements emitted by an output port.
`ssSetInputPortOptimOpts`	Specify the reusability of the memory allocated to the input port of an S-function.
`ssSetInputPortOverWritable`	Specify whether an input port is overwritable by an output port.

I/O Port — Signal Access (Continued)

Macro	Description
`ssSetOutputPortOptimOpts`	Specify the reusability of the memory allocated to the output port of an S-function.
`ssSetOutputPortOverwritesInputPort`	Specify whether an output port can share its memory buffer with an input port.

Model Reference

Macro	Description
`ssRTWGenIsModelReferenceRTWTarget`	Determine if the model reference Real-Time Workshop target is generating.
`ssRTWGenIsModelReferenceSIMTarget`	Determine if the model reference simulation target is generating.
`ssSetModelReferenceNormalModeSupport`	Specify if S-function can be used in referenced model simulating in normal mode.
`ssSetModelReferenceSampleTimeDefaultInheritance`	Specify that a submodel containing this S-function can inherit its sample time from its parent model.
`ssSetModelReferenceSampleTimeDisallowInheritance`	Specify that the use of this S-function in a submodel prevents the submodel from inheriting its sample time from its parent model.
`ssSetModelReferenceSampleTimeInheritanceRule`	Specify whether use of an S-function in a submodel prevents the submodel from inheriting its sample time from the parent model.

Run-Time Parameters

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

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

Sample Time

Macro	Description
ssGetInputPortSampleTime	Determine the sample time of an input port.
ssGetInputPortSampleTimeIndex	Get the sample time index of an input port.
ssGetNumSampleTimes	Get the number of sample times an S-function has.
ssGetOffsetTime	Determine one of an S-function's sample time offsets.
ssGetOutputPortSampleTime	Determine the sample time of an output port.
ssGetPortBasedSampleTimeBlockIsTriggered	Determine whether a block that uses port-based sample times resides in a triggered subsystem.
ssGetSampleTime	Determine one of an S-function's sample times.
ssGetTNext	Get the time of the next sample hit in a discrete S-function with a variable sample time.
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.
ssSetInputPortSampleTime	Set the sample time of an input port.

Macro	Description
`ssSetModelReferenceSampleTimeDefaultInheritancessSetModelReferenceSampleTi` `DefaultInheritance`	Specify that a submodel containing this S-function can inherit its sample time from its parent model.
`ssSetModelReferenceSampleTimeDisallowInheritancessSetModelReferenceSampleT` `DisallowInheritance`	Specify that the use of this S-function in a submodel prevents the submodel from inheriting its sample time from its parent model.
`ssSetModelReferenceSampleTimeInheritanceRulessSetModelReferenceSampleTime-` `InheritanceRule`	Specify whether use of an S-function in a submodel prevents the submodel from inheriting its sample time from the parent model.
`ssSetNumSampleTimes`	Set the number of sample times an S-function has.
`ssSetOffsetTime`	Specify the offset of a sample time.
`ssSetSampleTime`	Specify a sample time for an S-function.
`ssSetTNext`	Specify the time of the next sample hit in an S-function.

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.

Macro	Description
`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.
`ssGetSimStatus`	Determine the current simulation status.
`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.
`ssGetTNext`	Get the time of the next sample hit.
`ssGetTStart`	Get the start time of the current simulation.
`ssIsExternal`	Determine if the model is running in external mode.
`ssIsFirstInitCond`	Determine whether the current simulation time is equal to the simulation start time.

Macro	Description
`ssIsMajorTimeStep`	Determine whether the current time step is a major time step.
`ssIsMinorTimeStep`	Determine whether the current time step is a minor time step.
`ssIsVariableStepSolver`	Determine whether the current solver is a variable-step solver.
`ssRTWGenIsAccelerator`	Determine if the model is running in Accelerator mode.
`ssSetBlockReduction`	Request that Simulink attempt to reduce a block.
`ssSetSimStateCompliance`	Specify how Simulink treats an S-function when saving and restoring the simulation state of a model containing the S-function.
`ssSetSimStateVisibility`	Specify whether or not the simulation state of the S-function is visible (accessible) in the simulation state of the model.
`ssSetSolverNeedsReset`	Ask Simulink to reset the solver.
`ssSetStopRequested`	Ask Simulink to terminate the simulation at the end of the current time step.

State and Work Vector

Macro	Description
`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 DWork vector are real or complex numbers.
`ssGetDWorkDataType`	Get the data type of a DWork vector.
`ssGetDWorkName`	Get the name of a DWork vector.
`ssGetDWorkUsageType`	Determine how the DWork vector is used in S-function.
`ssGetDWorkUsedAsDState`	Determine whether a DWork vector is used as a discrete state vector.
`ssGetDWorkWidth`	Get the size of a DWork 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.
`ssGetNumContStates`	Determine the number of continuous states that an S-function has.

Macro	Description
`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.
`ssSetDWorkDataType`	Specify the data type of a data type work vector.

Macro	Description
`ssSetDWorkName`	Specify the name of a data type work vector.
`ssSetDWorkUsageType`	Specify how the DWork vector is used in S-function.
`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.

Code Generation

Macro	Description
`ssGetDWorkRTWIdentifier`	Get the identifier used to declare a DWork vector in code generated from the associated S-function.
`ssGetDWorkRTWIdentifierMustResolveToSignalObjectssGetDWorkRTWIdentifierMus` `ResolveToSignalObject`	Get a flag indicating if a DWork vector resolves to a `Simulink.Signal` object.
`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.
`ssGetPlacementGroup`	Get the name of the placement group of a block.
`ssRTWGenIsCodeGen`	Identify any code generation that is not used by the Accelerator.
`ssSetDWorkRTWIdentifier`	Set the identifier used to declare a DWork vector in code generated from the associated S-function.
`ssSetDWorkRTWIdentifierMustResolveToSignalObjectssSetDWorkRTWIdentifierMus` `ResolveToSignalObject`	Specify if a DWork vector resolves to a `Simulink.Signal` object.

Macro	Description
`ssSetDWorkRTWStorageClass`	Set the storage class of a DWork vector in code generated from the associated S-function.
`ssSetDWorkRTWTypeQualifier`	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.
`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.

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.

SimStruct Functions — Alphabetical List

ssGetDWorkRTWIdentifierMustResolveToSignalObject
ssGetDWorkRTWStorageClass
ssGetDWorkRTWTypeQualifier
ssGetDWorkUsageType
ssGetDWorkUsedAsDState
ssGetDWorkWidth
ssGetdX
ssGetErrorStatus
ssGetExplicitFCSSCtrl
ssGetFixedStepSize
ssGetInlineParameters
ssGetInputPortBufferDstPort
ssGetInputPortComplexSignal
ssGetInputPortConnected
ssGetInputPortDataType
ssGetInputPortDimensions
ssGetInputPortDimensionSize
ssGetInputPortDimensionsMode
ssGetInputPortDirectFeedThrough
ssGetInputPortFrameData
ssGetInputPortNumDimensions
ssGetInputPortOffsetTime
ssGetInputPortOptimOpts
ssGetInputPortOverWritable
ssGetInputPortRealSignal
ssGetInputPortRealSignalPtrs
ssGetInputPortRequiredContiguous
ssGetInputPortSampleTime
ssGetInputPortSampleTimeIndex
ssGetInputPortSignal
ssGetInputPortSignalPtrs
ssGetInputPortWidth
ssGetIWork
ssGetIWorkValue
ssGetModelName
ssGetModeVector
ssGetModeVectorValue

ssGetNonsampledZCs
ssGetNumContStates
ssGetNumDataTypes
ssGetNumDiscStates
ssGetNumDWork
ssGetNumInputPorts
ssGetNumIWork
ssGetNumModes
ssGetNumNonsampledZCs
ssGetNumOutputPorts
ssGetNumRunTimeParams
ssGetNumPWork
ssGetNumRWork
ssGetNumSampleTimes
ssGetNumSFcnParams
ssGetOffsetTime
ssGetOutputPortBeingMerged
ssGetOutputPortComplexSignal
ssGetOutputPortConnected
ssGetOutputPortDataType
ssGetOutputPortDimensions
ssGetOutputPortDimensionSize
ssGetOutputPortDimensionsMode
ssGetOutputPortFrameData
ssGetOutputPortNumDimensions
ssGetOutputPortOffsetTime
ssGetOutputPortOptimOpts
ssGetOutputPortRealSignal
ssGetOutputPortSampleTime
ssGetOutputPortSampleTimeIndex
ssGetOutputPortSignal
ssGetOutputPortWidth
ssGetParentSS
ssGetPath
ssGetPlacementGroup
ssGetPortBasedSampleTimeBlockIsTriggered
ssGetPWork

ssGetPWorkValue
ssGetRealDiscStates
ssGetRootSS
ssGetRunTimeParamInfo
ssGetRWork
ssGetRWorkValue
ssGetSampleTime
ssGetSFcnParam
ssGetSFcnParamsCount
ssGetSimMode
ssGetSimStatus
ssGetSolverMode
ssGetSolverName
ssGetStateAbsTol
ssGetStopRequested
ssGetT
ssGetTaskTime
ssGetTFinal
ssGetTNext
ssGetTStart
ssGetUserData
ssIsExternal
ssIsContinuousTask
ssIsFirstInitCond
ssIsMajorTimeStep
ssIsMinorTimeStep
ssIsSampleHit
ssIsSpecialSampleHit
ssIsVariableStepSolver
ssPrintf
ssPruneNDMatrixSingletonDims
ssRegDlgParamAsRunTimeParam
ssRegAllTunableParamsAsRunTimeParams
ssRegMdlSetInputPortDimensionsModeFcn
ssRegisterDataType
ssRegisterTypeFromNamedObject
ssRTWGenIsAccelerator

ssRTWGenIsCodeGen
ssRTWGenIsModelReferenceRTWTarget
ssRTWGenIsModelReferenceSimTarget
ssSampleAndOffsetAreTriggered
ssSampleAndOffsetAreTriggeredOrAsync
ssSetBlockReduction
ssSetCallSystemOutput
ssSetCurrentOutputPortDimensions
ssSetDataTypeSize
ssSetDataTypeZero
ssSetDWorkComplexSignal
ssSetDWorkDataType
ssSetDWorkName
ssSetDWorkRequireResetForSignalSize
ssSetDWorkRTWIdentifier
ssSetDWorkRTWIdentifierMustResolveToSignalObject
ssSetDWorkRTWStorageClass
ssSetDWorkRTWTypeQualifier
ssSetDWorkUsageType
ssSetDWorkUsedAsDState
ssSetDWorkWidth
ssSetErrorStatus
ssSetExplicitFCSSCtrl
ssSetExternalModeFcn
ssSetInputPortComplexSignal
ssSetInputPortDataType
ssSetInputPortDimensionInfo
ssSetInputPortDimensionsMode
ssSetInputPortDimsSameAsOutputPortDims
ssSetInputPortDirectFeedThrough
ssSetInputPortFrameData
ssSetInputPortMatrixDimensions
ssSetInputPortOffsetTime
ssSetInputPortOptimOpts
ssSetInputPortOverWritable
ssSetInputPortRequiredContiguous
ssSetInputPortSampleTime

ssSetInputPortVectorDimension
ssSetInputPortWidth
ssSetIWorkValue
ssSetModelReferenceNormalModeSupport
ssSetModelReferenceSampleTimeDefaultInheritance
ssSetModelReferenceSampleTimeDisallowInheritance
ssSetModelReferenceSampleTimeInheritanceRule
ssSetModeVectorValue
ssSetNumContStates
ssSetNumDiscStates
ssSetNumDWork
ssSetNumInputPorts
ssSetNumIWork
ssSetNumModes
ssSetNumNonsampledZCs
ssSetNumOutputPorts
ssSetNumPWork
ssSetNumRunTimeParams
ssSetNumRWork
ssSetNumSampleTimes
ssSetNumSFcnParams
ssSetOffsetTime
ssSetOneBasedIndexInputPort
ssSetOneBasedIndexOutputPort
ssSetOptions
ssSetOutputPortComplexSignal
ssSetOutputPortDataType
ssSetOutputPortDimensionInfo
ssSetOutputPortDimensionsMode
ssSetOutputPortFrameData
ssSetOutputPortMatrixDimensions
ssSetOutputPortOffsetTime
ssSetOutputPortOptimOpts
ssSetOutputPortOverwritesInputPort
ssSetOutputPortSampleTime
ssSetOutputPortVectorDimension
ssSetOutputPortWidth

ssSetPlacementGroup
ssSetPWorkValue
ssSetRWorkValue
ssSetRunTimeParamInfo
ssSetSampleTime
ssSetSFcnParamNotTunable
ssSetSFcnParamTunable
ssSetSignalSizesComputeType
ssSetSimStateCompliance
ssSetSimStateVisibility
ssSetSolverNeedsReset
ssSetStopRequested
ssSetTNext
ssSetUserData
ssSetVectorMode
ssSetZeroBasedIndexInputPort
ssSetZeroBasedIndexOutputPort
ssUpdateAllTunableParamsAsRunTimeParams
ssUpdateRunTimeParamData
ssUpdateDlgParamAsRunTimeParam
ssUpdateRunTimeParamInfo
ssWarning
ssWriteRTW2dMatParam
ssWriteRTWMx2dMatParam
ssWriteRTWMxVectParam
ssWriteRTWParameters
ssWriteRTWParamSettings
ssWriteRTWScalarParam
ssWriteRTWStr
ssWriteRTWStrParam
ssWriteRTWStrVectParam
ssWriteRTWVectParam
ssWriteRTWWorkVect

ssAddOutputDimsDependencyRule

Purpose

Register a method to handle current dimensions update.

Syntax

```
void ssAddOutputDimsDependencyRule(SimStruct *S, int_T outIdx,
DimsDependInfo_T ruleInfo)
```

Arguments

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

`outIdx`
: Output port index.

`ruleInfo`
: Structure containing the dimensions propagation method information.

Returns

No return value

Description

Use this function in `mdlSetWorkWidths`mdl to register a method that updates the dimensions for the output port when there is a change in the current input signal dimensions. The method is for the case when the output signal size depends only on the input signal size. It is called only when a dimensions update is necessary, instead of calling in each sim loop pass.

Requires you to set up ruleinfo struct, which includes

- int *inputs — Index to inputs whose dimensions affect the output dimensions
- int numInputs — Number of inputs that affect the output dimensions
- SetOutputDimsFcn SetOutputDimsFcn — Function to update the output dimensions based on the input dimensions.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/`

`sfun_varsize_holdStatesUntilReset.c`.

ssAddVariableSizeSignalsRuntimeChecker

Purpose

Register a method to check the current input dimensions

Syntax

```
void ssAddVariableSizeSignalsRuntimeChecker (SimStruct *S, RuntimeChecker
```

Arguments

`S`
SimStruct representing an S-Function block.

`type`
Enum value corresponding to the checker type.

Returns

No return value

Description

Use this function in mdlSetWorkWidths to register a method for checking the input dimensions. Two types of checks are allowed INPUTS_DIMS_MATCH INPUTS_DISALLOW_EMPTY_SIGNAL The input dimensions matching checks that the current dimensions of all the input ports match, otherwise an error is thrown. The empty signal checker verifies that no input signal is empty at runtime, otherwise an error is thrown.

Languages

C, C++

Purpose Enable S-function to work with multidimensional signals

Syntax

```
void ssAllowSignalsWithMoreThan2D(SimStruct *S)
```

Arguments

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

Description Allows S-function to use multidimensional signals. You must call the `ssAllowSignalsWithMoreThan2D` function from the `mdlInitializeSizes` function.

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/srcsfun_matadd.c` used in `sfcndemo_matadd.mdl`.

See Also `mdlInitializeSizes`

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.

`fcn`
: External mode function.

Returns An error string on failure, and `NULL` if successful.

Description Invokes the external mode function for `S`. This macro is for internal use. User-written S-functions should not use the `ssCallExternalModeFcn` macro.

Languages C, C++

See Also `ssSetExternalModeFcn`

Purpose

Call the update and outputs methods of a function-call subsystem

Syntax

`int_T ssCallSystemWithTid(SimStruct *S, int_T element, int_T tid)`

Arguments

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

`element`
: Index of the output port element corresponding to the function-call subsystem.

`tid`
: Task ID.

Returns

An `int_T 1` if successful; otherwise, `0`.

Description

Use in `mdlOutputs` to call the update and outputs methods of a function-call subsystem connected to the S-function. The invoking syntax is

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

See “Function-Call Subsystems” on page 8-60 for more information on how to use `ssCallSystemWithTid`.

Note `ssCallSystemWithTid` can only be used with function-call subsystems that either hold or inherit their states, based on the setting of the **States when enabling parameter** of the function-call subsystem’s Trigger block. If the function-call subsystem needs to have its states reset upon enabling, the S-function must explicitly enable and disable the function-call subsystem using `ssSetExplicitFCSSCtrl`, `ssEnableSystemWithTid` and `ssDisableSystemWithTid`.

Languages C, C++

Example See the S-function *matlabroot*`toolbox/simulink/simdemos/simfeatures/src/sfun_fcncall.c`.

See Also `ssSetCallSystemOutput`, `ssDisableSystemWithTid`, `ssEnableSystemWithTid`

Purpose

Disable a function-call subsystem connected to this S-function block

Syntax

`int_T ssDisableSystemWithTid(SimStruct *S, int_T element, int_T tid)`

Arguments

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

`element`
: Index of the output port element corresponding to the function-call subsystem.

`tid`
: Task ID.

Returns

An `int_T` `1` if successful; otherwise, `0`.

Description

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

```
if (!ssDisableSystemWithTid(S, element, tid)) {
  /* Error occurred which will be reported by the Simulink engine*/
  return;
}
```

Note Before invoking this function, the S-function must have specified that it explicitly enables and disables the function-call subsystems that it calls. See `ssSetExplicitFCSSCtrl` and `ssEnableSystemWithTid` for more information. If the S-function has not done this, invoking `ssDisableSystemWithTid` results in an error.

This function resets the outputs of any Outport blocks in the function-call subsystem whose **Outputs when disabled** parameter is set to `reset`.

Languages

C, C++

Example See the example in the reference page for `ssEnableSystemWithTid`.

See Also `ssCallSystemWithTid`, `ssEnableSystemWithTid`, `ssSetExplicitFCSSCtrl`

Purpose

Enable a function-call subsystem connected to this S-function

Syntax

```
int_T ssEnableSystemWithTid(SimStruct *S, int_T element, int_T tid)
```

Arguments

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

`element`
: Index of the output port element corresponding to the function-call subsystem.

`tid`
: Task ID.

Returns

An `int_T` `1` if successful; otherwise, `0`.

Description

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

```
if (!ssEnableSystemWithTid(S, element, tid)) {
  /* Error occurred which will be reported by the Simulink engine*/
  return;
}
```

Note Before invoking this function, the S-function must have specified that it explicitly enables and disables the function-call subsystems that it calls. See `ssSetExplicitFCSSCtrl` for more information. If the S-function has not done this, invoking `ssEnableSystemWithTid` results in an error.

The effect of invoking this function depends on the setting of the **States when enabling parameter** of the function-call subsystem's Trigger block. If the parameter is set to `reset`, this function invokes the function-call subsystem's initialize method and then its enable method. The subsystem's initialize and enable methods in turn invoke the initialize and enable methods of any blocks in the subsystem that have these methods. Initialize methods reset the states of blocks that have states, e.g., Integrator blocks, to their initial values. Thus, if the Trigger block's **States when enabling** option is set to `reset`, invoking this function effectively resets the states of the function-call subsystem. If the Trigger block's **States when enabling** option is set to `held`, this function simply invokes the subsystem's enable method, without invoking its initialize method and hence without resetting its states.

Languages

C, C++

Example

This example shows how to configure an S-function to reset the states of the function-call subsystem it calls. The code below shows the macros needed in two callbacks. The `mdlInitializeSampleTimes` callback first specifies that the S-function explicitly enables and disables the function-call subsystem. The `mdlOutputs` callback then handles the actual enabling and disabling of the function-call subsystem in order to reset the states.

The following code in `mdlInitializeSampleTimes` uses `ssSetExplicitFCSSCtrl` to enable the S-function to explicitly enable and disable the function-call subsystem. `ssSetCallSystemOutput` then specifies that the function-call subsystem is invoked by the first element of the S-function's first output port.

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, 0.1);
    ssSetOffsetTime(S, 0, 0.0);

    /* Explicitly enable/disable function-call subsystem */
    ssSetExplicitFCSSCtrl(S,1);

    /* Call function-call subsystem on first element */
    ssSetCallSystemOutput(S,0);

    ssSetModelReferenceSampleTimeDefaultInheritance(S);
} /* End mdlInitializeSampleTimes */
```

The `mdlOutputs` callback is shown below. It first uses `ssEnableSystemWithTid` to enable the function-call subsystem at the beginning of the simulation. The function-call subsystem must be enabled before it can be called using `ssCallSystemWithTid`. After the simulation has run for 10 seconds, the `mdlOutputs` callback invokes `ssDisableSystemWithTid` to disable the function-call subsystem. By invoking `ssEnableSystemWithTid` again, the function-call subsystem is re-enabled and the states are reset.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T  *x = ssGetRealDiscStates(S);
    real_T  *y = ssGetOutputPortRealSignal(S,1);
    time_T   t = ssGetT(S); /* Simulation time */

    /* Enable function-call subsystem at start of simulation */
    if (t==0) {
        if (!ssEnableSystemWithTid(S,0,tid)) {
            return;
        }
    }

    /* Call function-call subsystem */
    if (!ssCallSystemWithTid(S,0,tid)) {
        return;
    }

    /* Disable/re-enable function-call subsystem when time = 10 */
    if (t==10) {
        if (!ssDisableSystemWithTid(S,0,tid)) {
            return;
        }
        if (!ssEnableSystemWithTid(S,0,tid)) {
            return;
        }
        }
        y[0] = x[0];

    } /* End mdlOutputs */
```

See Also

ssCallSystemWithTid, ssDisableSystemWithTid, ssSetExplicitFCSSCtrl

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.

Returns

A pointer (`real_T *`) to a array containing the tolerance for each continuous state.

Description

Use in `mdlStart` to get the absolute tolerances used by the variable-step solver for this simulation.

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`. Invoking `ssGetAbsTol` when using a fixed-step solver results in an error.

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 the S-function *matlabroot*`toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl` for a complete example that uses this function.

See Also `ssGetStateAbsTol`, `ssIsVariableStepSolver`

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.

Returns

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 `1` if the block has previously requested that it be reduced. When invoked after the simulation loop has begun, this macro returns `1` if the block has actually been reduced, i.e., eliminated from the list of blocks to be executed during the simulation loop. Otherwise, returns `0`.

Description

Use to determine if a block requested block reduction, or to determine if the block has already been reduced.

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, the Simulink engine invokes the `mdlTerminate` method only if the block has set its `SS_OPTION_CALL_TERMINATE_ON_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_ON_EXIT` option before the simulation loop has begun and invoke `ssGetBlockReduction` in its `mdlTerminate` method.

Languages

C, C++

See Also

`ssSetBlockReduction`

Purpose

Get a block's continuous states

Syntax

`real_T *ssGetContStates(SimStruct *S)`

Arguments

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

Returns

A pointer (`real_T *`) to the continuous state vector as an array of length `ssGetNumContStates(S)`. Returns `NULL` if the S-function does not have any continuous states.

Description

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

Languages

C, C++

Example

The following lines from the file *`matlabroot`*`toolbox/simulink/simdemos/simfeatures/src/csfunc.c` show how to initialize the continuous states in `mdlInitializeConditions` and calculate the state derivatives in `mdlDerivatives`. This S-function is used in the model `sfcndemo_csfunc.mdl`.

```
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *x0 = ssGetContStates(S);
    int_T lp;

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

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[1][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);
}
```

See Also

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

ssGetCurrentInputPortDimensions

Purpose Gets the current size of dimension dIdx of input port pIdx

Syntax

```
Int_T ssGetCurrentInputPortDimensions(SimStruct *S,int_T pIdx, int_T dIdx
```

Arguments

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

`pIdx`
: Input port index being polled.

`dIdx`
: Index of dimension being polled.

Returns An int_T value indicating the size of dimension dIdx.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_concat1`

Purpose Gets the total width (total number of elements) of the signal at input port pIdx

Syntax

```
Int_T ssGetCurrentInputPortWidth(SimStruct *S,int_T pIdx)
```

Arguments

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

`pIdx`
: Input port index being polled.

Returns An int_T value indicating the current width of the signal at input port pIdx.

Languages C, C++

ssGetCurrentOutputPortDimensions

Purpose Gets the current size of dimension dIdx of the signal at output port pIdx.

Syntax

```
Int_T ssGetCurrentOutputPortDimensions(SimStruct *S,int_T pIdx, int_T dIc
```

Arguments

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

`pIdx`
: Output port index being polled.

`dIdx`
: Index of dimension being polled.

Returns An int_T value indicating the size of dimension dIdx.

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_holdSta`

Purpose

Gets the total width (total number of elements) of the signal at output port pIdx.

Syntax

```
Int_T ssGetCurrentOutputPortWidth(SimStruct *S,int_T pIdx)
```

Arguments

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

`pIdx`
: Output port index being polled.

Returns

An int_T value indicating the current width of the signal at output port pIdx.

Languages

C, C++

ssGetDataTypeId

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.

Returns

The ID of the custom data type specified by `name` if `name` is a registered type name. Otherwise, returns `INVALID_DTYPE_ID` and reports an error.

Description

Use to obtain the data type ID of a custom data type.

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

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages

C, 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`

Purpose

Get the built-in data type associated with a data type alias

Syntax

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

Arguments

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

id
: ID of the data type alias.

Returns

The built-in data type ID associated with the data type alias specified by `id`.

Description

Use to obtain the built-in data type associated with a data type alias. For a list of built-in values for the data type ID `DtypeId`, see `ssGetInputPortDataType`.

Note `ssGetDataTypeIdAliasedThruTo` requires that you use `ssSetOptions` to set the `SS_OPTION_SUPPORTS_ALIAS_DATA_TYPES` option in order for your to S-function recognize data type aliases.

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages

C, C++

See Also

`Simulink.AliasType`, `ssRegisterTypeFromNamedObject`

Purpose

Get the name of a data type

Syntax

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

Arguments

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

`id`
: ID of a data type.

Returns

The name of the data type specified by `id`, if `id` is valid. Otherwise, returns `NULL` and reports an error.

Description

Use to obtain the name of a data type.

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

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages

C, C++

Example

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

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

See Also

`ssRegisterDataType`

Purpose

Get the size of a custom data type

Syntax

```
int_T ssGetDataTypeSize(SimStruct *S, DTypeId id)
```

Arguments

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

`id`
: ID of a data type.

Returns

An `int_T` value indicating the size of the data type specified by `id`, if `id` is valid and the data type's size has been set. Otherwise, returns `INVALID_DTYPE_SIZE` and reports an error.

Description

Use to obtain the size of a custom data type.

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

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages

C, 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 a data type.

Returns

A pointer (`void *`) 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, returns `NULL` and reports an error.

Description

Use to obtain the zero representation of a data type.

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

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages

C, 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.

Returns

A pointer (`real_T *`) to the discrete state vector as an array of length `ssGetNumDiscStates(S)`. Returns `NULL` if the S-function does not have any discrete states.

Description

Use to obtain the block's discrete states. 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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/dsfunc.c` used in `sfcndemo_dsfunc.mdl` for an example on how to initialize and update the discrete state vector.

See Also

`ssGetNumDiscStates`, `ssGetRealDiscStates`, `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.

Returns

The data type ID of an S-function parameter represented by a MATLAB array. Returns `INVALID_DTYPE_ID` if the MATLAB data type does not map to any built-in Simulink data type ID.

Description

This function returns an enumerated type representing the data type. The enumerated type `DTypeId` is defined in `simstruc_types.h`. The following table shows the equivalency of Simulink, MATLAB, and C data types.

Simulink Data Type DTypeId	Simulink Data Type DTypeId Index	MATLAB Data Type mxClassID	C Data Type
`SS_DOUBLE`	0	`mxDOUBLE_CLASS`	`real_T`
`SS_SINGLE`	1	`mxSINGLE_CLASS`	`real32_T`
`SS_INT8`	2	`mxINT8_CLASS`	`int8_T`
`SS_UINT8`	3	`mxUINT8_CLASS`	`uint8_T`
`SS_INT16`	4	`mxINT16_CLASS`	`int16_T`
`SS_UINT16`	5	`mxUINT16_CLASS`	`uint16_T`
`SS_INT32`	6	`mxINT32_CLASS`	`int32_T`
`SS_UINT32`	7	`mxUINT32_CLASS`	`uint32_T`
`SS_BOOLEAN`	8	`mxUINT8_CLASS+ logical`	`boolean_T`

If a MATLAB data type, for example `mxSTRUCT_CLASS`, does not map to any Simulink data type, the return value is `INVALID_DTYPE_ID`.

Otherwise the return value is one of the `enum` values in `BuiltInDTypeId`. For example, for `mxUINT16_CLASS`, the return value is `SS_UINT16`.

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

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_param.` used in `sfcndemo_dtype_param.mdl` to learn how to use data typed parameters in an S-function.

See Also

`ssGetSFcnParam`

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)-1`.

Returns

A pointer (`void *`) to the DWork vector specified by the index `vector`.

Description

Use to obtain a pointer to a particular DWork vector. See "How to Use DWork Vectors" on page 7-7 for more information on DWork vectors.

Note Do not use any of the `SimStruct` functions that get information about DWork vector if your S-function does not contain any DWork vectors. Otherwise, your S-function produces errors during simulation.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl` to learn how to use DWork vectors in an S-function.

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)-1`.

Returns

`COMPLEX_YES` (1) if the specified vector contains complex numbers; otherwise, `COMPLEX_NO` (0).

Description

Use to determine the numeric type of the DWork vector specified by the index `vector`.

Languages

C, C++

Example

The following example throws an error if the first DWork vector is not complex.

```
CSignal_T cs = ssGetDWorkComplexSignal(S, 0);
if(cs == COMPLEX_NO) {
     ssSetErrorStatus(S,"Signal must be complex.");
}
```

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)-1`.

Returns

The data type ID of the DWork vector specified by the index `vector` if the data type is valid. Otherwise, returns `INVALID_DTYPE_ID`.

Description

Use to obtain the data type of a particular DWork vector.
For a list of built-in values for the data type ID `DTypeId`, see `ssGetInputPortDataType`.

Languages

C, C++

Example

The following example checks the data type of the first DWork vector.

```
DTypeId dt = ssgetDWorkDataType(S, 0);
if(dt == INVALID_DTYPE_ID) {
     ssSetErrorStatus(S,"Invalid data type.");
}
```

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)-1`.

Returns

A pointer (`char_T *`) to the string representing the name of the DWork vector specified by the index `vector`.

Description

Use to obtain the name of a particular DWork vector.

Languages

C, C++

Example

The following lines get the DWork vector name and display it at the MATLAB command prompt.

```
char_T *dname = ssGetDWorkName(S, 0);
ssPrintf("Initial value of %s has been set to 1\n",dname);
```

See Also

`ssSetDWorkName`

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_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)-1`.

Returns

A pointer (`char_T *`) to the string used as the Real-Time Workshop identifier for the DWork vector specified by the index `vector`. Returns `NULL` if no Real-Time Workshop identifier is specified.

Description

Use to obtain the identifier used in code generated by the Real-Time Workshop product to declare the DWork vector specified by `vector`.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl` to learn how to use DWork vectors in an S-function.

See Also

`ssSetDWorkRTWIdentifier`

ssGetDWorkRTWIdentifierMustResolveToSignalObject

Purpose

Get a flag indicating if a DWork vector resolves to a `Simulink.Signal` object

Syntax

```
unsigned int_T ssGetDWorkRTWIdentifierMustResolveToSignalObject(
 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)-1`.

Returns

An `unsigned int_T` flag with the value of `0`, `1`, or `2`.

Description

Use this function to determine if the DWork vector specified by `vector` resolves to a `Simulink.Signal` object. The output value indicates the following.

- `0` indicates the Simulink engine tries to resolve the DWork vector to a `Simulink.Signal` object. In this mode, the engine only tries to resolve the DWork vector to a `Simulink.Signal` object if implicit signal resolution is enabled. The Data Validity parameter **Signal resolution** on the Diagnostics pane of the Configuration Parameters dialog box controls implicit signal resolution. When this option is set to `Explicit only`, the engine interprets a flag of `0` as it would a flag of `2`. See the "Signal resolution" reference page in *Simulink Graphical User Interface* for more information on implicit signal resolution.
- `1` declares that the DWork vector must resolve to a `Simulink.Signal` object. The engine invokes an error if it cannot resolve the DWork vector to a `Simulink.Signal` object.
- `2` indicates the engine does not try to resolve the DWork vector to a `Simulink.Signal` object.

Languages

C, C++

See Also `ssSetDWorkRTWIdentifierMustResolveToSignalObject`

Purpose

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

Syntax

```
ssRTWStorageType ssGetDWorkRTWStorageClass(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)-1`.

Returns

One of the enumerated types defined for `ssRTWStorageType`. Permissible values are:

```
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

Use to obtain the storage class of the DWork vector specified by `vector`. The storage class is a code-generation attribute that determines how code generated by the Real-Time Workshop product for this S-function allocates memory for this work vector (see “Signal Storage Concepts” in the *Real-Time Workshop User’s Guide*).

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_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)-1`.

Returns

A pointer (`char_T *`) to a string indicating the C type qualifier used to declare the DWork vector specified by the index `vector`. Returns `NULL` if no type qualifier is specified.

Description

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

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl` to learn how to use DWork vectors in an S-function.

See Also

`ssSetDWorkRTWTypeQualifier`

Purpose

Determine how DWork vector is used in S-function

Syntax

```
ssDWorkUsageType ssGetDWorkUsageType(SimStruct *S, int_T vector)
```

Arguments

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

`vector`
: Index of the data type work vector.

Returns

One of the enumerated types defined for `ssDWorkUsageType` in *`matlabroot`*`/simulink/include/simstruc_types.h`. Permissible values are:

- `SS_DWORK_USED_AS_DWORK`
- `SS_DWORK_USED_AS_DSTATE`
- `SS_DWORK_USED_AS_SCRATCH`
- `SS_DWORK_USED_AS_MODE`

Description

Use this macro to determine how the DWork vector specified by the `vector` is being used in the S-function. By default, a DWork vector returns `SS_DWORK_USED_AS_DWORK`.

Languages

C, C++

Example

For more information on using DWork vectors, see Chapter 7, "Using Work Vectors".

See Also

`ssSetDWorkUsageType`

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)-1`.

Returns

The `int_T` value `1` (`SS_DWORK_USED_AS_DSTATE`) if this vector is used to store a block's discrete states. Otherwise, returns `0` (`SS_DWORK_USED_AS_DWORK`).

Description

Use to determine if the DWork vector specified by the index `vector` is used to store discrete state values.

Languages

C, C++

Example

For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

See Also

`ssSetDWorkUsedAsDState`

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)-1`.

Returns

The `int_T` number of elements in the DWork vector specified by the index `vector`.

Description

Use to obtain the size of a particular DWork vector.

Languages

C, C++

Example

For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

See Also

`ssSetDWorkWidth`

ssGetdX

Purpose

Get the derivatives of a block's continuous states

Syntax

`real_T *ssGetdX(SimStruct *S)`

Arguments

`S`

SimStruct representing an S-Function block or Simulink model.

Returns

A pointer (`real_T *`) to an array containing the derivatives of the continuous states of `S`, which can be a block or the model. Returns `NULL` if there are no continuous states.

Description

Use this macro in `mdlDerivatives` to get the derivatives of a model or block's continuous states. Use `ssGetNumContStates(S)` to get the length of the array.

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

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/csfunc.c` used in `sfcndemo_csfunc.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl` for examples using this function.

See Also

`ssGetNumContStates`, `ssGetContStates`

ssGetErrorStatus

Purpose Get a string that identifies the last error

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

Arguments `S`

SimStruct representing an S-Function block.

Returns A pointer (`char_T *`) to a string that identifies the last error message.

Description Use to determine the last error.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_errhdl.c` used in `sfcndemo_sfun_errhdl.mdl`.

See Also `ssSetErrorStatus`

ssGetExplicitFCSSCtrl

Purpose

Determine whether this S-function explicitly enables and disables the function-call subsystems that it invokes

Syntax

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

Arguments

`S`

SimStruct representing an S-Function block.

Returns

The `unsigned int_T 1` if `S` explicitly enables or disables the function-control subsystem that it invokes. Otherwise, returns `0`.

Description

Use to determine if this S-function explicitly enables and disables the function-call subsystems that it invokes. See "Function-Call Subsystems" on page 8-60 for more information on interfacing S-functions and function-call subsystems.

Languages

C, C++

See Also

`ssSetExplicitFCSSCtrl`, `ssEnableSystemWithTid`, `ssDisableSystemWithTid`

Purpose

Get the fixed step size of the model containing the S-function.

Syntax

```
time_T ssGetFixedStepSize(SimStruct *S)
```

Arguments

`S`
    SimStruct representing an S-Function block.

Returns

A `time_T` value indicating the fixed step size of the model containing the S-function if the model is configured to use a fixed-step solver. Otherwise returns `0`.

Description

Use this macro in methods called after the compilation phase, i.e., in `mdlSetWorkWidths` or later, to obtain the fixed step size of the model containing the S-function. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

Languages

C, C++

Example

The following lines get and display the fixed step size.

```
time_T fss = ssGetFixedStepSize(S);
ssPrintf("Fixed step size : %g\n",fss);
```

See Also

`ssIsVariableStepSolver`

ssGetInlineParameters

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.

Returns A Boolean value `true` or `false` depending on the setting for the **Inline parameters** optimization option.

Description Returns `true` if the user has checked the **Inline parameters** option on the **Optimization** pane of the Configuration Parameters dialog box (see the "Optimization Pane" reference page in *Simulink Graphical User Interface*).

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_port_constant.c` used in `sfcndemo_port_constant.mdl`.

Purpose

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

Syntax

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

Arguments

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

`inputPortIdx`
: Index of an input port on `S`

Returns

The `int_T` index of the output port that reuses the memory buffer of the input port indicated by the index `inputPortIdx`. If none of the S-function's output ports reuse this input port buffer, returns `INVALID_PORT_IDX (-1)`.

Description

Use this function any time after model initialization to get the index of the output port that reuses the specified input port's buffer.

During model compilation, the Simulink engine may allocate the same memory buffer to the specified input port and an output port of this S-function if the following conditions apply:

- Neither the input port nor the output port are test points.
- The input port is overwritable (`ssSetInputPortOverWritable`).

Languages

C, C++

See Also

`ssSetNumInputPorts`, `ssSetInputPortOverWritable`

ssGetInputPortComplexSignal

Purpose Determine whether an input port accepts complex signals.

Syntax `CSignal_T ssGetInputPortComplexSignal(SimStruct *S,input_T port)`

Arguments

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

`port`
: Index of an input port.

Returns `COMPLEX_YES` (1) if `port` accepts complex signals, `COMPLEX_NO` (0) if `port` does not accept complex signals, and `COMPLEX_INHERITED` (-1) if `port` inherits its numeric type from the port to which it is connected.

Description Use to obtain the numeric type of the input port specified by the index `port`.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_cplx.c` used in `sfcndemo_cplx.mdl`.

See Also `ssSetInputPortComplexSignal`

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.

`port`
: Port whose connection status is needed.

Returns

Either an `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) value indicating if the input port specified by the index `port` is connected to a nonvirtual block.

Description

Returns `1` (`true`) if the specified `port` on the block represented by `S` is connected directly or indirectly, i.e., via virtual blocks, 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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dynsize.c` used in `sfcndemo_sfun_dynsize.mdl`.

See Also

`ssGetOutputPortConnected`, `ssSetNumInputPorts`

Purpose

Get the data type of an input port

Syntax

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

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

Returns

The data type ID of the input port specified by `port`. Returns `DYNAMICALLY_TYPED` if the input port inherits its data type.

Description

Use to obtain the data type of a particular input port. The file *`matlabroot`*`/simulink/include/simstruc_types.h` defines the list of built-in data types associated with the index `DTypeId` as follows.

Integer Data Type ID (DTypeId)	Built-in Data Type
0	SS_DOUBLE
1	SS_SINGLE
2	SS_INT8
3	SS_UINT8
4	SS_INT16
5	SS_UINT16
6	SS_INT32
7	SS_UINT32
8	SS_BOOLEAN

Languages

C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime3.c` used in `sfcndemo_runtime.mdl`.

See Also `ssSetInputPortDataType`

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.

Returns A pointer (`int_T *`) to an array of integers. The array contains elements with the value `DYNAMICALLY_SIZED` (`-1`) when the size of a dimension is unknown.

Description Use to obtain the dimensions of the signal accepted by the input port with index `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, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_frmdft.c` used in `sfcndemo_frame.mdl`. Running this model requires a Signal Processing Blockset license.

See Also `ssGetInputPortNumDimensions`

Purpose

Get the size of one dimension of the signal entering an input port

Syntax

```
int_T ssGetInputPortDimensionSize(SimStruct *S, int_T port, int_T dIdx
```

Arguments

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

`port`
: Index of the output port.

`dIdx`
: Index of the dimension.

Returns

An `int_T` value indicating the size of dimension, `dIdx`, at the input port specified by `port`. Returns `1` if the `dIdx` is greater than or equal to the number of input port dimensions.

Description

Use to obtain the size of one dimension if a particular input port.

Languages

C, C++

See Also

`ssGetInputPortDimensions`

ssGetInputPortDimensionsMode

Purpose Gets the dimensions mode of the input port indexed by pIdx

Syntax

```
DimensionsMode_T ssGetInputPortDimensionsMode(SimStruct *S,int_T pIdx)
```

Arguments

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

`pIdx`
: Input port index being polled.

Returns A DimensionsMode_T value indicating the current dimensions mode. Possible values are INHERIT_DIMS_MODE FIXED_DIMS_MODE and VARIABLE_DIMS_MODE

Languages C, C++

Purpose

Determine whether a port has direct feedthrough

Syntax

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

Arguments

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

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

Returns

The `int_T` value `1` if the input port specified by the index `port` has direct feedthrough. Otherwise, returns `0`.

Description

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

Languages

C, C++

See Also

`ssSetInputPortDirectFeedThrough`

ssGetInputPortFrameData

Purpose

Determine whether a port accepts signal frames

Syntax

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

Arguments

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

`port`
: Index of an input port.

Returns

A value of type `Frame_T`, indicating if the input port specified by the index `port` accepts signal frames.

Description

Use to determine if a particular input port accepts frame-based signals. Possible return values include:

- `FRAME_INHERITED`

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

  Port accepts unframed input only.
- `FRAME_YES`

  Port accepts frame input only.

Languages

C, 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.

Returns

A positive integer indicating the number of dimensions of the input port specified by the index `port`, or `DYNAMICALLY_SIZED`, if the number of dimensions is unknown.

Description

Use to obtain the number of dimensions of a particular input port.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl`.

See Also

`ssGetInputPortDimensions`

ssGetInputPortOffsetTime

Purpose Get the offset time of an input port

Syntax

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

Arguments

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

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

Returns A `real_T` value indicating the offset time of the input port specified by the index `inputPortIdx`.

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

Languages C, C++

See Also `ssSetInputPortOffsetTime`, `ssGetInputPortSampleTime`

Purpose

Get the reusability setting of the memory allocated to the input port of an S-function

Syntax

```
uint_T ssGetInputPortOptimOpts(SimStruct *S, int_T port)
```

Arguments

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

`port`
: Index of an input port of `S`.

Returns

One of the following values:

- `SS_NOT_REUSABLE_AND_GLOBAL`
- `SS_REUSABLE_AND_LOCAL`
- `SS_REUSABLE_AND_GLOBAL`
- `SS_NOT_REUSABLE_AND_LOCAL`

Description

Use this macro to get the reusability of an S-function input port. For more information about the possible return values, see `ssSetInputPortOptimOpts`.

Languages

C, C++

See Also

`ssSetInputPortOptimOpts`

Purpose Determine whether an input port can be overwritten

Syntax

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

Arguments

`S`
: SimStruct representing an S-function block.

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

Returns An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) value indicating if the input port specified by the index `port` can be overwritten.

Description Use to determine if a particular input port can be overwritten. Returns `1` (`true`) if the input port can be overwritten.

Languages C, C++

See Also `ssSetInputPortOverWritable`

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

Syntax

```
const real_T *ssGetInputPortRealSignal(SimStruct *S, int_T inputPortIc
```

Arguments

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

`inputPortIdx`
: Index of the port whose signal is required.

Returns A pointer (`real_T *`) to a real signal on the input port specified by the index `inputPortIdx`.

Description Use to obtain the real signal on a particular 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`.

Note The `ssGetInputPortRealSignal` macro becomes a function when you compile your S-function in debug mode (`mex -g`).

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 the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_frmdft.c` used in `sfcndemo_frame.mdl` for a complete example using this function. Running this model requires a Signal Processing Blockset license.

See Also `ssSetInputPortRequiredContiguous`, `ssGetInputPortSignal`, `mdlInitializeSizes`

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 the port whose signal is required.

Returns

Pointers to the elements of a signal of type `double` connected to the input port specified by the index `port`.

Description

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. The input port index starts at 0 and ends at the number of input ports minus 1.

Note The `ssGetInputPortRealSignalPtrs` macro becomes a function when you compile your S-function in debug mode (`mex -g`).

Languages

C, 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 =
                ssGetInputPortRealSignalPtrs(S,i);
  int_T nu = ssGetInputPortWidth(S,i);
  for (j = 0; j < nu; j++) {
    SomeFunctionToUseInputSignalElement(*uPtrs[j]);
  }
}
```

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl` for a complete example using this function.

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.

`port`
: Index of an input port.

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) value indicating if the signal elements entering a port must be contiguous.

Description

Use to determine if the signal elements entering a port must be contiguous. Returns `1` (`true`) if the signal elements entering the specified port must occupy contiguous areas of memory. Otherwise, returns `0` (`false`). 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++

Example

See the reference page for `ssGetInputPortSignal` for an example that uses this function.

See Also

`ssSetInputPortRequiredContiguous`, `ssGetInputPortSignal`, `ssGetInputPortSignalPtrs`

Purpose Get the sample time of an input port

Syntax

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

Arguments

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

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

Returns A `real_T` value indicating the sample time of the input port specified by the index `inputPortIdx`.

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. Use `ssGetSampleTime` to determine the sample time of S-functions that do not use port-based sample times.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also `ssSetInputPortSampleTime`, `ssGetInputPortOffsetTime`

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.

`inputPortIdx`
: Index of the input port whose sample time index is to be returned.

Returns

An `int_T` value indicating the sample time index of the input port specified by the index `inputPortIdx`. Returns `CONSTANT_TID` (`-2`) for constant (`inf`) sample times.

Description

Use in any routine after sample time propagation (i.e., in or after `mdlInitializeSampleTimes`) to determine the sample time index of an input port. You should use this macro only if you have specified port-based sample times.

The Simulink engine returns the sample time index in the context of the S-function's sample times, not the entire model's sample times. Consequently, the sample time index (`sti`) returned by `ssGetInputPortSampleTimeIndex` and the task ID (`tid`) passed into the S-function by the Simulink engine are not equivalent. Use the index returned by `ssGetInputPortSampleTimeIndex` in calls to `ssIsSampleHit`, etc., to determine if the S-function is running at one of its sample rates. For example, the following `mdlOutputs` method checks if the sample time index for the first input port is executing.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
   int sti = ssGetInputPortSampleTimeIndex(S,0);
   if (ssIsSampleHit(S, sti, tid)) {
         CodeForThisSampleRateHere()
   }
}
```

The Simulink engine returns an index of `CONSTANT_TID` (`-2`) for constant (`inf`) sample times. In this case, the sample time index and model-wide task ID are identical.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_port_constant.c` used in `sfcndemo_port_constant.mdl`.

See Also

`ssSetInputPortSampleTime`

Purpose

Get the address of a contiguous signal entering an input port

Syntax

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

Arguments

`S`
SimStruct representing an S-Function block.

`inputPortIdx`
Index of the port whose address is required.

Returns

A pointer (`void *`) to the input port specified by the index `inputPortIdx`.

Description

Use to get the address of a contiguous signal entering an input port. Your S-function should use this macro only if `mdlInitializeSizes` has specified that the elements of the input signal be contiguous, using `ssSetInputPortRequiredContiguous`. For non-contiguous input, use the `ssGetInputPortSignalPtrs` function.

Note The `ssGetInputPortSignal` macro becomes a function when you compile your S-function in debug mode (`mex -g`).

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`, `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.

Returns

Pointer to an array of signal element pointers for the specified input port.

Description

Use to obtain pointers to an input port's signal elements. 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`).

Use this function only for non-contiguous input. If you have contiguous input, use the `ssGetInputPortSignal` function.

Note The `ssGetInputPortSignalPtrs` macro becomes a function when you compile your S-function in debug mode (`mex -g`).

Languages

C, 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  uPtrs = (InputInt8PtrsType)u;
int_T              nu    = ssGetInputPortWidth(S,i);
```

```
for (j = 0; j < nu; j++) {
    /* uPtrs[j] is an int8_T pointer that points to the j-th
       element of the input signal.
     */
 UseInputInSomeFunction(*uPtrs[j]);
}
```

See the S-function *matlabroot*//toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c used in sfcndemo_dtype_io.mdl for a complete example that uses this function.

See Also

ssGetInputPortRealSignalPtrs, ssGetInputPortSignal

Purpose Get the width of an input port

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

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of the port whose width is required.

Returns An `int_T` value indicating the number of elements in the input signal. If the number of elements is unknown, returns `DYNAMICALLY_SIZED`.

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 C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc_sat.c` used in `sfcndemo_sfun_zc_sat.mdl`.

See Also `ssSetInputPortWidth`

ssGetIWork

Purpose Get a block's integer work vector

Syntax `int_T *ssGetIWork(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns A pointer (`int_T *`) to the integer work vector for this S-function.

Description Use to access 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, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl`.

See Also `ssGetNumIWork`, `ssSetIWorkValue`, `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.

Returns

The `int_T` value stored in the `idx` element of the integer work vector for this S-function. Returns `NULL` if no value was assigned into the `idx` element of the IWork vector.

Description

Use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to obtain an element of the IWork vector. 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`.

Languages

C, C++

Example

The following statement

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

is equivalent to

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

For a complete example using `ssGetIWork`, see the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl`.

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

Purpose

Get the model name

Syntax

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

Arguments

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

Returns

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

Description

Get the name of an S-function or Simulink model.

Languages

C, C++

See Also

`ssGetPath`

ssGetModeVector

Purpose

Get the mode vector

Syntax

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

Arguments

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

Returns

A pointer (`int_T *`) to the mode vector.

Description

Use to obtain a pointer 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++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc.c` used in `sfcndemo_sfun_zc.mdl`.

See Also

`ssSetNumModes`

Purpose Get an element of a block's mode vector

Syntax `int_T ssGetModeVectorValue(SimStruct *S, int_T elementx)`

Arguments

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

`elementx`
: Index of a mode vector element.

Returns An `int_T` value for the element of the mode vector specified by the index `elementx`. Returns `NULL` if no value was assigned into the `elementx` element of the mode vector.

Description Use to obtain the specified mode vector element, where the element index start at 0 and end at the total number of modes minus 1.

Languages C, C++

Example The following statement

```
int_T v = ssGetModeVectorValue(S, 0);
```

is equivalent to

```
int_T* wv = ssGetModeVector(S);
int_T v = wv[0];
```

See Also `ssSetModeVectorValue`, `ssGetModeVector`

ssGetNonsampledZCs

Purpose Get the zero-crossing signal values

Syntax `real_T *ssGetNonsampledZCs(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns A pointer (`real_T *`) to the zero-crossing signal values.

Description Use to obtain 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)`.

Languages C, C++

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];
      }
  }
```

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc_sat.c` used in `sfcndemo_sfun_zc_sat.mdl` for a complete example that uses this function.

See Also

`ssGetNumNonsampledZCs`

ssGetNumContStates

Purpose Get the number of continuous states that a block has

Syntax `int_T ssGetNumContStates(SimStruct *S)`

Arguments `S`

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

Returns An `int_T` value indicating the number of continuous states.

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

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl`.

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

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.

Returns

An `int_T` value indicating the number of registered data types.

Description

Use to obtain 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. For a list of built-in data types, see `BuiltInDTypeId` in `simstruc_types.h`.

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 the Simulink engine invokes the `mdlInitializeSizes` callback function.

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages

C, C++

See Also

`ssRegisterDataType`

ssGetNumDiscStates

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 or a Simulink model.

Returns An `int_T` value indicating the number of discrete states.

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

Languages C, C++

Example See the S-function *`matlabroot`*`/simulink/src/stvdtf.c` used in `sfcndemo_stvdtf.mdl`.

See Also `ssSetNumDiscStates`, `ssGetNumContStates`

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.

Returns

An `int_T` value indicating the number of DWork vectors in this S-function.

Description

Use to obtain the number of data type work vectors used by `S`.

Languages

C, C++

See Also

`ssSetNumDWork`

ssGetNumInputPorts

Purpose Get the number of input ports that a block has

Syntax `int_T ssGetNumInputPorts(SimStruct *S)`

Arguments

`S`
: SimStruct representing an S-function block.

Returns An `int_T` value indicating the number of input ports.

Description Use in any routine (except `mdlInitializeSizes`) to determine how many input ports an S-Function block has.

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_multiport.c` used in `sfcndemo_sfun_multiport.mdl` and the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl`.

See Also `ssGetNumOutputPorts`

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.

Returns An `int_T` value indicating the size of the IWork vector.

Description Use this macro in any routine except `mdlInitializeSizes` to obtain the size of the integer (`int_T`) work vector used by the block represented by `S`.

Languages C, C++

See Also `ssSetNumIWork`, `ssGetNumRWork`

ssGetNumModes

Purpose Get the size of the mode vector

Syntax `int_T ssGetNumModes(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns An `int_T` value indicating the size of the mode work vector.

Description Use this macro in any routine except `mdlInitializeSizes` to obtain the size of the mode work vector.

Languages C, C++

See Also `ssSetNumNonsampledZCs`, `ssGetNonsampledZCs`

Purpose Get the size of the zero-crossing vector

Syntax `int_T ssGetNumNonsampledZCs(SimStruct *S)`

Arguments

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

Returns An `int_T` value indicating the size of the zero-crossing vector.

Description Use this macro in any routine except `mdlInitializeSizes` to obtain the size of the zero-crossing vector.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc_sat.c` used in `sfcndemo_sfun_zc_sat.mdl`.

See Also `ssSetNumNonsampledZCs`, `ssGetNonsampledZCs`

ssGetNumOutputPorts

Purpose

Get the number of output ports that a block has

Syntax

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

Arguments

`S`

SimStruct representing an S-Function block.

Returns

An `int_T` value indicating the number of output ports.

Description

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

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_multiport.c` used in `sfcndemo_sfun_multiport.mdl`.

See Also

`ssGetNumInputPorts`

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.

Returns

An `int_T` value indicating the number of run-time parameters.

Description

Use this function to get the number of run-time parameters created by this S-function. The number of run-time parameters is specified using `ssSetNumRunTimeParams`. Run-time parameters are registered using `ssRegDlgParamAsRunTimeParam` or `ssSetRunTimeParamInfo`. See "Run-Time Parameters" on page 8-8 for more information.

Languages

C, C++

See Also

`ssSetNumRunTimeParams`, `ssRegDlgParamAsRunTimeParam`, `ssSetRunTimeParamInfo`

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.

Returns An `int_T` value indicating the size of the PWork vector.

Description Use this macro in any routine except `mdlInitializeSizes` to obtain the size of the pointer work vector used by the block represented by `S`.

Languages C, 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.

Returns An `int_T` value indicating the size of the RWork vector.

Description Use this macro in any routine except `mdlInitializeSizes` to obtain the size of the floating-point (`real_T`) work vector used by the block represented by `S`.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfunmem.c` used in `sfcndemo_sfunmem.mdl`.

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.

Returns

An `int_T` value indicating the total number of port-based and block-based sample times. Returns `-1` if the block has unspecified port-based sample times.

Description

Use in any routine (except `mdlInitializeSizes`) to determine the number of sample times used by `S`.

Languages

C, C++

See Also

`ssSetNumSampleTimes`

ssGetNumSFcnParams

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.

Returns An `int_T` value indicating the number of expected S-function parameters.

Description Use to determine the number of parameters that `S` expects the user to enter.

Tip Use `ssGetSFcnParamsCount` to determine the number of parameters the user actually entered for `S`.

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl` and the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/csfunc.c` used in `sfcndemo_csfunc.mdl`.

See Also `ssSetNumSFcnParams` `ssGetSFcnParamsCount`

ssGetOffsetTime

Purpose Get one of an S-function's sample time offsets.

Syntax

```
time_T ssGetOffsetTime(SimStruct *S, int_T sti);
```

Arguments

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

`sti`
: Index of sample time offset to be returned

Returns A `time_T` value indicating the sample time offset.

Description Use to obtain the sample time offset of `S` corresponding to the index `sti`. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc_sat.c` used in `sfcndemo_sfun_zc_sat.mdl`.

See Also `ssSetOffsetTime`, `ssGetSampleTime`

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.

`port`
: Index of the output port.

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) value indicating if this output port signal is being merged with other signals.

Description

Use this macro in and after the `mdlSetWorkWidths` method. Returns `1(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).

The output port must be made reusable using `ssSetOutputPortOptimOpts` to be connected to a Merge block.

Languages

C, C++

See Also

`mdlSetWorkWidths`, `ssSetOutputPortOptimOpts`

ssGetOutputPortComplexSignal

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.

Returns

`COMPLEX_YES` (1) if `port` accepts complex signals, `COMPLEX_NO` (0) if `port` does not accept complex signals, and `COMPLEX_INHERITED` (-1) if `port` inherits its numeric type from the port to which it is connected.

Description

Use to obtain the numeric type of the output port specified by the index `port`.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_cplx.c` used in `sfcndemo_cplx.mdl`.

See Also

`ssSetOutputPortComplexSignal`

Purpose

Determine whether an output port is connected to a nonvirtual block

Syntax

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

Arguments

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

`port`
: Port whose connection status is needed.

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) value indicating if this output port signal is connected to a nonvirtual block.

Description

Use anywhere except in `mdlInitializeSizes` or `mdlCheckParameters`. Returns `1` (`true`) if the specified output `port` on the block represented by `S` is connected directly or indirectly, i.e., via virtual blocks, to a nonvirtual block. The S-function must have previously set the number of output ports in `mdlInitializeSizes`, using `ssSetNumOutputPorts`.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_dynsize.c` used in `sfcndemo_sfun_dynsize.mdl`.

See Also

`ssGetInputPortConnected`, `ssSetNumOutputPorts`

ssGetOutputPortDataType

Purpose

Get the data type of an output port

Syntax

```
DTypeId ssGetOutputPortDataType(SimStruct *S, int_T port)
```

Arguments

`S`
: SimStruct representing an S-function block.

`port`
: Index of an output port.

Returns

The data type ID of the output port specified by the index `port`. Returns `DYNAMICALLY_TYPED` if the output port inherits its data type.

Description

Use to obtain the data type ID of a particular output port. The file *`matlabroot`*`/simulink/include/simstruc_types.h` defines the list of built-in data types associated with the index `DTypeId` as follows.

Integer Data Type ID (DTypeId)	Built-in Data Type
0	SS_DOUBLE
1	SS_SINGLE
2	SS_INT8
3	SS_UINT8
4	SS_INT16
5	SS_UINT16
6	SS_INT32
7	SS_UINT32
8	SS_BOOLEAN

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl`.

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.

Returns A pointer (`int_T *`) to an array of integers. The array contains elements with the value `DYNAMICALLY_SIZED` (`-1`) when the size of a dimension is unknown.

Description Use to obtain the dimensions of the signal leaving the output port specified by the index `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, C++

See Also `ssGetOutputPortNumDimensions`

Purpose

Get the size of one dimension of the signal leaving an output port

Syntax

```
int_T ssGetOutputPortDimensionSize(SimStruct *S, int_T port,
 int_T dIdx)
```

Arguments

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

`port`
: Index of the output port.

`dIdx`
: Index of the dimension.

Returns

An `int_T` value indicating the size of dimension, `dIdx`, at the output port specified by `port`. Returns `1` if the `dIdx` is greater than or equal to the number of output port dimensions.

Description

Use to obtain the size of one dimension of a particular output port.

Languages

C, C++

See Also

`ssGetOutputPortDimensions`

ssGetOutputPortDimensionsMode

Purpose

Gets the dimensions mode of the output port indexed by pIdx

Syntax

```
DimensionsMode_T ssGetOUTputPortDimensionsMode(SimStruct *S,int_T pIdx)
```

Arguments

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

`pIdx`
: Output Port index being polled.

Returns

A DimensionsMode_T value indicating the current dimensions mode. Possible values are INHERIT_DIMS_MODE FIXED_DIMS_MODE and VARIABLE_DIMS_MODE

Languages

C, C++

Purpose

Determine whether a port outputs signal frames

Syntax

```
Frame_T ssGetOutputPortFrameData(SimStruct *S,  int_T port)
```

Arguments

`S`
: SimStruct representing an S-function block.

`port`
: Index of an output port.

Returns

A value of type `Frame_T`, indicating if the output port specified by the index `port` produces signal frames.

Description

Use to determine if a particular output port produces frame-based signals. Possible return values include:

- `FRAME_INHERITED`

  Port outputs either frame or unframed data.
- `FRAME_NO`

  Port outputs unframed data only.
- `FRAME_YES`

  Port outputs frame data only.

Languages

C, C++

See Also

`ssSetOutputPortFrameData`

ssGetOutputPortNumDimensions

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.

Returns

A positive integer indicating the number of dimensions of the output port specified by the index `port`, or `DYNAMICALLY_SIZED`, if the number of dimensions is unknown.

Description

Use to determine the number of dimensions of output port specified by the index `port`.

Languages

C, C++

Example

The following lines return the length of the last dimension of the first output port.

```
int_T numDim = ssGetOutputPortNumDimensions(S, 0);
int_T lDim - ssGetOutputPortDimensionSize(S, 0, numDim-1);
```

See Also

`ssGetOutputPortDimensionSize`

Purpose Get the offset time of an output port

Syntax

```
real_T ssGetOutputPortOffsetTime(SimStruct *S,
 int_T outputPortIdx)
```

Arguments

`S`
    SimStruct representing an S-function block.

`outputPortIdx`
    Index of an output port.

Returns A `real_T` value indicating the offset time of the output port specified by the index `outputPortIdx`.

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

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also `ssSetOutputPortOffsetTime`, `ssGetOutputPortSampleTime`

Purpose

Get the reusability setting of the memory allocated to the output port of an S-function

Syntax

```
uint_T ssGetOutputPortOptimOpts(SimStruct *S, int_T port)
```

Arguments

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

`port`
: Index of an output port of `S`.

Returns

One of the following values:

- `SS_NOT_REUSABLE_AND_GLOBAL`
- `SS_REUSABLE_AND_LOCAL`
- `SS_REUSABLE_AND_GLOBAL`
- `SS_NOT_REUSABLE_AND_LOCAL`

Description

Use this macro to get the reusability of an S-function output port. For more information about these values, see `ssSetOutputPortOptimOpts`.

Languages

C, C++

See Also

`ssSetOutputPortOptimOpts`

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.

Returns

A contiguous `real_T` vector of length equal to the width of the 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.

Note You cannot use `ssGetOutputPortRealSignal` anywhere except in `mdlOutputs` if you have specified that the output ports are reusable using `ssSetOutputPortOptimOpts`. For example, if the outputs have been specified as reusable with the `SS_REUSABLE_AND_LOCAL` flag, the `mdlUpdate` routine errors out when it tries to access output memory that is unavailable.

Languages

C, C++

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();
    }
  }
```

See the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl` for a complete example that uses this function.

See Also

`ssGetInputPortRealSignalPtrs`

ssGetOutputPortSampleTime

Purpose

Get the sample time of an output port

Syntax

```
real_T ssGetOutputPortSampleTime(SimStruct *S, int_T outputPortIdx)
```

Arguments

`S`

SimStruct representing an S-Function block.

`outputPortIdx`

Index of an output port.

Returns

A `real_T` value indicating the sample time of the output port specified by the index `outputPortIdx`.

Description

Use in any routine (except `mdlInitializeSizes`) to determine the sample time of an output port. You should use this macro only if you have specified the sample times as port-based. Use `ssGetSampleTime` to determine the sample time of S-functions that do not use port-based sample times.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also

`ssSetOutputPortSampleTime`, `ssGetOutputPortOffsetTime`

ssGetOutputPortSampleTimeIndex

Purpose

Get sample time index of output port

Syntax

```
int_T ssGetOutputPortSampleTimeIndex(SimStruct *S,
 int_T outputPortIdx)
```

Arguments

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

`outputPortIdx`
: Index of the output port whose sample time index is to be returned.

Returns

An `int_T` value indicating the sample time index for the output port specified by the index `outputPortIdx`. Returns `CONSTANT_TID` (`-2`) for constant (`inf`) sample times.

Description

Use in any routine after sample time propagation (i.e., in or after `mdlInitializeSampleTimes`) to determine the sample time index of an output port. You should use this macro only if you have specified port-based sample times.

The Simulink engine returns the sample time index in the context of the S-function's sample times, not the entire model's sample times. Consequently, the sample time index (`sti`) returned by `ssGetOutputPortSampleTimeIndex` and the task ID (`tid`) passed into the S-function by the Simulink engine are not equivalent. Use the index returned by `ssGetOutputPortSampleTimeIndex` in calls to `ssIsSampleHit`, etc., to determine if the S-function is running at one of its sample rates. For example, the following `mdlOutputs` method uses the sample time index to check if the first output port is executing.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
   int sti = ssGetOutputPortSampleTimeIndex(S,0);
   if (ssIsSampleHit(S, sti, tid)) {
          CodeForThisSampleRateHere()
   }
}
```

The Simulink engine returns an index of `CONSTANT_TID` (`-2`) for constant (`inf`) sample times. In this case, the sample time index and model-wide task ID are identical.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also

`ssSetOutputPortSampleTime`, `ssGetOutputPortSampleTime`

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.

Returns

A pointer (`void *`) to the vector of signal elements output at the port specified by the index `port`.

Description

Use in any simulation loop routine, `mdlInitializeConditions`, or `mdlStart` to obtain a vector of signal elements emitted by an output port.

Note You cannot use `ssGetOutputPortSignal` anywhere except in `mdlOutputs` if you have specified that the output ports are reusable using `ssSetOutputPortOptimOpts`. For example, if the outputs have been specified as reusable with the `SS_REUSABLE_AND_LOCAL` flag, the `mdlUpdate` routine errors out when it tries to access output memory that is unavailable.

Note If the port outputs a signal of type `double` (`real_T`), use `ssGetOutputPortRealSignal` to get the signal vector and avoid the need to type cast the output of `ssGetOutputPortSignal`.

Languages

C, C++

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]);
 }
}
```

See the S-function *`matlabroot`*`/simulink/src/sfun_port_constant.c` used in `sfcndemo_port_constant.mdl` for a complete example that uses this function.

See Also

`ssGetOutputPortRealSignal`

ssGetOutputPortWidth

Purpose

Get the width of an output port

Syntax

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

Arguments

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

`port`
: Index of an output port.

Returns

An `int_T` value indicating the width of the output port specified by the index `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

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc.c` used in `sfcndemo_sfun_zc.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl`.

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.

Returns

The parent `SimStruct` of `S`, or `NULL` if `S` is the root `SimStruct`.

Description

Use to obtain the parent of a `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. This macro is for internal use. User-written S-functions should not use the `ssGetParentSS` macro.

Languages

C, C++

See Also

`ssGetRootSS`

ssGetPath

Purpose Get the path of a block

Syntax `const char_T *ssGetPath(SimStruct *S)`

Arguments `S`

SimStruct representing an S-function block or a Simulink model.

Returns A pointer (`char_T *`) to a string containing the path to a block.

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 the MATLAB command prompt and is not part of a simulation.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multiport.c` used in `sfcndemo_sfun_multiport.mdl`.

See Also `ssGetModelName`

Purpose

Get the name of the placement group of a block

Syntax

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

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).

Returns

A pointer (`char_T *`) to the string indicting the name of the S-function's placement group.

Description

Use this macro in `mdlInitializeSizes` to get the name of this block's placement group. A placement group is an advanced feature for S-functions that are either a source block (i.e., a block without input ports) or a sink block (i.e., a block without output ports). All S-functions with the same placement group will be placed adjacent to each other in the sorted list. There is no correlation between different placement groups.

Note This macro is typically used to create device driver blocks.

Languages

C, 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.

Returns The Boolean value `true` if `S` uses port-based sample times and resides in a triggered subsystem. Otherwise, returns `false`.

Description Use this macro in `mdlOutputs` and `mdlUpdate` to decode whether to use the block's triggered or nontriggered 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, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_port_triggered.` used in `sfcndemo_port_triggered.mdl`.

See Also `ssSampleAndOffsetAreTriggered`

Purpose Get a block's pointer work vector

Syntax `void **ssGetPWork(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns A pointer to the PWork vector.

Description Use to access 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, C++

See Also `ssGetNumPWork`, `ssGetPWorkValue`

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.

Returns

The (`void *`) element of the PWork vector at the index `idx`.

Description

Use to access an element of 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. A return value of `NULL` indicates that no value was assigned into the `idx` element of the pointer work vector.

Languages

C, C++

Example

The following statement

```
void* v = ssGetPWorkValue(S, 0);
```

is equivalent to

```
void** wv = ssGetPWork(S);
void* v = wv[0];
```

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.

Returns

The discrete state vector as an array of `real_T` elements of length `ssGetNumDiscStates(S)`.

Description

Same as `ssGetDiscStates`.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/dsfunc.c` used in `sfcndemo_dsfunc.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/vsfunc.c` used in `sfcndemo_vsfunc.mdl`.

See Also

`ssGetDiscStates`

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.

Returns The `SimStruct` at the root of the `SimStruct` hierarchy.

Description Use to obtain the root of the `SimStruct` hierarchy containing `S`. If `S` is, itself, the root `SimStruct` then `ssGetRootSS` returns `S`.

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. This macro is for internal use. User-written S-functions should not use the `ssGetRootSS` macro.

Languages C, C++

See Also `ssGetParentSS`

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.

Returns

A pointer to the `ssParamRec` describing the attributes of the run-time parameter specified by the index `param`.

Description

Use to obtain 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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multiport.c` used in `sfcndemo_sfun_multiport.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime4.c` used in `sfcndemo_runtime.mdl`.

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.

Returns

A pointer (`real_T *`) to the RWork vector.

Description

Use to access 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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dynsize.c` used in `sfcndemo_sfun_dynsize.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfunmem.c` used in the model `sfcndemo_sfunmem.mdl`.

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.

Returns

The `real_T` value stored in the RWork vector element specified by the index `idx`.

Description

Use to obtain the `idx` element of 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.

Languages

C, C++

Example

The following statement

```
real_T v = ssGetRWorkValue(S, 0);
```

is equivalent to

```
real_T* wv = ssGetRWork(S);
real_T v = wv[0];
```

For complete examples using `ssGetRWork`, see the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dynsize.c` used in `sfcndemo_sfun_dynsize.mdl` and the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfunmem.c` used in `sfcndemo_sfunmem.mdl`.

See Also

`ssGetNumRWork`, `ssGetRWork`, `ssSetRWorkValue`

Purpose

Get one of an S-function's sample times.

Syntax

```
time_T ssGetSampleTime(SimStruct *S, int_T sti);
```

Arguments

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

`sti`
: Index of the sample time to be returned.

Returns

A `time_T` value indicating the sample time associated with the index `sti`.

Description

Use to obtain the sample time of `S` corresponding to the sample time index `sti`. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

Note You cannot call this macro in S-functions that only use port-based sample times. You should use the macros `ssGetInputPortSampleTime` and `ssGetOutputPortSampleTime`.

Languages

C, C++

Example

The following statement

```
time_T t = ssGetSampleTime(S, 0);
```

returns the S-function's current sample time.

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc_sat.c` used in `sfcndemo_sfun_zc_sat.mdl` for a complete example that uses this function.

See Also

`ssSetSampleTime`

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.

Returns

A pointer (`const mxArray *`) to the value of the S-function parameter specified by `index`.

Description

Use in any routine to access a parameter entered in the S-Function Block Parameters dialog box, where *`index`* starts at 0 and is less than `ssGetSFcnParamsCount(S)`.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl`.

See Also

`ssGetSFcnParamsCount`

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.

Returns

An `int_T` value.

Description

Use to query the number of parameters that a user entered into the Block Parameters dialog box for the S-Function block represented by `S`.

Tip Use `ssGetNumSFcnParams` to obtain the number of parameters `S` expects

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl` and the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/csfunc.c` used in `sfcndemo_csfunc.mdl`.

See Also

`ssGetNumSFcnParams`

ssGetSimMode

Purpose

Get the simulation mode of an S-Function block

Syntax

`SS_SimMode ssGetSimMode(SimStruct *S)`

Arguments

`S`

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

Returns

An enumerated value of type `SS_SimMode`.

Description

Use to query the simulation mode of the block represented by `S`. Possible return values are:

- `SS_SIMMODE_NORMAL`

  Running a normal Simulink simulation
- `SS_SIMMODE_SIZES_CALL_ONLY`

  Invoked by the editor to obtain the number of ports
- `SS_SIMMODE_RTWGEN`

  Generating code regardless of the simulation mode
- `SS_SIMMODE_EXTERNAL`

  Running an external mode simulation

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/vsfunc.c` used in `sfcndemo_vsfunc.mdl`.

See Also

`ssRTWGenIsModelReferenceRTWTarget`, `ssRTWGenIsModelReferenceSimTarget`, `ssRTWGenIsCodeGen`, `ssRTWGenIsAccelerator`, `ssIsExternal`, `ssGetSolverName`

Purpose

Get the current simulation status of an S-Function block

Syntax

```
ssGetSimStatus(SimStruct *S, SS_SimStatus *status)
```

Arguments

`S`

SimStruct representing an S-Function block.

`status`

`SS_SimStatus` object that returns the current simulation status.

Description

Determines the simulation status of the block represented by `S` and stores the value in the `SS_SimStatus` object `status`. The variable `status` can have one of the following values, defined in `simstruc_types.h`:

- `SIMSTATUS_STOPPED`

  The simulation has terminated
- `SIMSTATUS_UPDATING`

  The Simulink engine is updating the model
- `SIMSTATUS_INITIALIZING`

  The simulation is initializing
- `SIMSTATUS_RUNNING`

  The simulation is running
- `SIMSTATUS_PAUSED`

  The simulation is paused
- `SIMSTATUS_TERMINATING`

  The simulation is terminating
- `SIMSTATUS_EXTERNAL`

  The simulation is running in external mode

Languages C, C++

Example The following lines obtain the current simulation status:

```
SS_SimStatus status;        // Define the return variable
ssGetSimStatus(S, &status);
```

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.

Returns

One of

- `SOLVER_MODE_AUTO`
- `SOLVER_MODE_SINGLETASKING`
- `SOLVER_MODE_MULTITASKING`

Description

Use this macro to get the solver mode for this S-function. 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++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_frmunbuff.c` used in `sfcndemo_frame.mdl` for an example that uses this function. Running this model requires a Signal Processing Blockset license.

See Also

`ssGetSimMode`, `ssIsVariableStepSolver`

ssGetSolverName

Purpose

Get the name of the solver being used to solve the S-function

Syntax

```
char_T *ssGetSolverName(SimStruct *S)
```

Arguments

`S`

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

Returns

A pointer (`char_T *`) to the name of the solver being used.

Description

Use in any routine after `mdlInitializeSampleTimes` to return the name of the solver being used to solve the S-function or model represented by `S`.

Languages

C, C++

See Also

`ssGetSimMode`, `ssIsVariableStepSolver`

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.

`state`
: Index of the state whose absolute tolerance is to be returned.

Returns

A `real_T` value for the absolute tolerance of the state referenced by the index `state`.

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++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl`.

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.

Returns The `int_T` value of the simulation stop requested flag.

Description Gets the value of the simulation stop requested flag. If the value is `1`, the Simulink engine halts the simulation at the end of the current time step.

Languages C, C++

See Also `ssSetStopRequested`

Purpose

Get the current simulation time

Syntax

```
time_T ssGetT(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Returns

A value of type `time_T` indicating the current simulation time.

Description

Use to determine 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. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

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

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/vsfunc.c` used in `sfcndemo_vsfunc.mdl`.

See Also

`ssGetTaskTime`, `ssGetTStart`, `ssGetTFinal`

ssGetTaskTime

Purpose

Get the current time for the current task

Syntax

`time_T 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.

Returns

A value of type `time_T`.

Description

Use to determine 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. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

The `ssGetTaskTime` macro should be called only inside an `ssIsSampleHit` check. It will not give the correct results if called with the `tid` passed into `mdlOutputs`.

Languages

C, C++

Example

The following example illustrates a correct usage of this macro:

```
static void mdlOutputs( SimStruct *S, int_T tid )
{
 double t;
 if(ssIsSampleHit(S,0,tid)) {
  t = ssGetTaskTime(S,0);
  ssPrintf("Task 0 sample hit in %s time = %g\n",
     ssGetPath(S),t);
 }
 if(ssIsSampleHit(S,1,tid)) {
  t = ssGetTaskTime(S,1);
```

```
    ssPrintf("Task 1 sample hit in %s time = %g\n",
       ssGetPath(S),t);
   }
  }
```

See Also `ssGetT`

ssGetTFinal

Purpose Get the simulation stop time

Syntax `time_T ssGetTFinal(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns A value of type `time_T`.

Description Use to query the stop time of the current simulation. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

Languages C, C++

See Also `ssGetT`, `ssGetTStart`

Purpose Get the time of the next sample hit

Syntax `time_T ssGetTNext(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns A value of type `time_T`.

Description Use to determine the next time that a sample hit occurs in a discrete S-function with a variable sample time. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

Languages C, C++

See Also `ssSetTNext`, `mdlGetTimeOfNextVarHit`

ssGetTStart

Purpose Get the simulation start time

Syntax `time_T ssGetTStart(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns A value of type `time_T`.

Description Use to query the start time of the current simulation. See *`matlabroot`*`/extern/include/tmwtypes.h` for a description of the `time_T` data type.

Languages C, C++

See Also `ssGetT`, `ssGetTFinal`

Purpose Access user data

Syntax `void *ssGetUserData(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Returns A pointer (`void *`) to the S-function's user data.

Description Use to access the user data associated with this block. See `ssSetUserData` for more information on specifying user data.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multiport.c` used in `sfcndemo_sfun_multiport.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime4.c` used in `sfcndemo_runtime.mdl`.

See Also `ssSetUserData`

Purpose Determine if the model is running in external mode.

Syntax

```
boolean_T ssIsExternal(SimStruct *S)
```

Arguments

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

Returns The Boolean value `true` if the model is running in external mode. Returns `false` otherwise. Note: if ssGetSimMode returns “external”, the result is identical to that of ssIsExternal.

Description Use this macro in an S-function to determine whether the model is currently running in external mode.

Languages C, C++

See Also `ssRTWGenIsModelReferenceRTWTarget`, `ssRTWGenIsModelReferenceSimTarget`, `ssRTWGenIsCodeGen`, `ssRTWGenIsAccelerator`, `ssGetSimMode`

Purpose

Determine whether a task is continuous

Syntax

```
boolean_T ssIsContinuousTask(SimStruct *S, int_T tid)
```

Arguments

`S`
SimStruct representing an S-Function block.

`tid`
Task ID.

Returns

The Boolean value `true` when the simulation is executing the continuous task. Otherwise, returns `false`.

Description

Use in `mdlOutputs` or `mdlUpdate` when your S-function has multiple sample times to determine if the task represented by the task ID `tid` is the continuous task. For example:

```
if (ssIsContinuousTask(S, tid)) {
    /* Executing in the continuous task */
    if (ssIsSpecialSampleHit(S, 1, 0, tid)) {
        real_T *zoh = ssGetRWork(S);
        real_T *xC  = ssGetContStates(S);
        *zoh = *xC;
    }
}
```

You should not use this in single-rate S-functions, or if you did not register a continuous sample time.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/mixedm.c` used in `sfcndemo_mixedm.mdl`.

See Also

`ssSetSampleTime`, `ssIsSpecialSampleHit`

ssIsFirstInitCond

Purpose

Determine whether the simulation time is equal to the start time.

Syntax

```
boolean_T ssIsFirstInitCond(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Returns

The Boolean value `true` if the current simulation time is equal to the simulation start time. Otherwise, returns `false`.

Note If this S-function resides in an enabled subsystem configured to reset states, and if the enabled system is enabled at the start time, then `mdlInitializeConditions` is called a second time and the Boolean value of `ssIsFirstInitCond` is `true`.

Description

Use inside `mdlInitializeConditions` to determine if the callback is being invoked at the start of the simulation.

Languages

C, C++

See Also

`mdlInitializeConditions`

Purpose Determine whether the simulation is in a major step

Syntax

```
boolean_T ssIsMajorTimeStep(SimStruct *S)
```

Arguments

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

Returns The Boolean value `true` if the simulation is in a major time step. Otherwise, returns `false`.

Description Use to determine if the simulation is currently running at a major or minor time step.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc.c` used in `sfcndemo_sfun_zc.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl`.

See Also `ssIsMinorTimeStep`

ssIsMinorTimeStep

Purpose

Determine whether the simulation is in a minor step

Syntax

```
boolean_T ssIsMinorTimeStep(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Returns

The Boolean value `true` if the simulation is in a minor time step. Otherwise, returns `false`.

Description

Use to determine if the simulation is currently running at a minor or major time step.

Languages

C, C++

See Also

`ssIsMajorTimeStep`

Purpose

Determine whether the sample time is hit

Syntax

```
boolean_T ssIsSampleHit(SimStruct *S, int_T st_index, int_T tid)
```

Arguments

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

`st_index`
: Index of the sample time.

`tid`
: Task ID.

Returns

The Boolean value `true` when the simulation is executing in the task represented by task ID `tid`. Otherwise, returns `false`.

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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/mixedmex.c` used in `sfcndemo_mixedmex.mdl`.

See Also

`ssIsContinuousTask`, `ssIsSpecialSampleHit`

ssIsSpecialSampleHit

Purpose Determine whether the sample time is hit

Syntax

```
boolean_T ssIsSpecialSampleHit(SimStruct *S, int_T sti1,
 int_T sti2, int_T tid)
```

Arguments

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

`sti1`
: Index of the sample time.

`sti2`
: Index of the sample time.

`tid`
: Task ID.

Returns The Boolean value `true` if a sample hit has occurred at `sti1` and a sample hit has also occurred at `sti2` in the same time step. Otherwise, returns `false`.

Description 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 8-47.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/mixedm.c` used in `sfcndemo_mixedm.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also `ssIsSampleHit`

Purpose

Determine if a variable-step solver is being used to solve the S-function

Syntax

```
boolean_T ssIsVariableStepSolver(SimStruct *S)
```

Arguments

`S`

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

Returns

The Boolean value `true` if the solver being used to solve `S` is a variable-step solver. Otherwise, returns `false`.

Description

Use to determine if the simulation is being solved using a variable or fixed-step solver. This is useful when you are creating S-functions that have zero crossings and an inherited sample time.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/vsfunc.c` used in `sfcndemo_vsfunc.mdl` and the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl`.

See Also

`ssGetSimMode`, `ssGetSolverName`

Purpose

Print a variable-content message

Syntax

`int_T ssPrintf(const char_T *msg, ...)`

Arguments

`msg`

Message. Must be a ANSI®[2] C `printf`-style string with optional variable replacement parameters.

`...`

Optional replacement arguments.

Returns

A positive value indicating the number of bytes transmitted. Returns a negative number indicating an error.

Description

Prints a variable-content `msg`. This macro expands to `mexPrintf` when the S-function is compiled via `mex` for use in a Simulink simulation. When the S-function is compiled for use with the Real-Time Workshop code generation, 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 code generation, you can avoid a call altogether, using the `SS_STDIO_AVAILABLE` macro defined by `simstruc.h`. For example:

```
#if defined(SS_STDIO_AVAILABLE)
 ssPrintf("my message ...");
#endif
```

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl`.

See Also

`ssWarning`

2. ANSI is a registered trademark of the American National Standards Institute, Inc.

Purpose

Prune trailing singleton dimensions

Syntax

```
void ssPruneNDMatrixSingletonDims(SimStruct *S, DimsInfo_T *dimsInfo)
```

Arguments

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

`dimsInfo`
: Structure of type `DimsInfo_T` that specifies the dimensionality of the signals accepted by `port`.

  See `ssSetInputPortDimensionInfo` for a description of this structure.

Description

Removes all trailing singleton dimensions in the `dimsInfo` structure. This function modifies the number of dimensions in the `dimsInfo` structure, but it does not change the dimension size array in `dimsInfo`.

Languages

C, C++

See Also

`ssSetInputPortDimensionInfo`

ssRegDlgParamAsRunTimeParam

Purpose

Register a dialog parameter as a run-time parameter

Syntax

```
void ssRegDlgParamAsRunTimeParam(SimStruct *S, int_T dlgIdx,
 int_T rtIdx, const char_T *name, DTypeId dtId)
```

Arguments

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

`dlgIdx`
: Index of the dialog parameter.

`rtIdx`
: Index of the run-time parameter.

`name`
: Name of the run-time 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. For a list of built-in values for the data type ID `dtId`, see `ssGetInputPortDataType`.

If the data type conversion results in precision loss or data overflow, the Simulink engine takes the action defined by the **Diagnostics Pane: Data Validity** configuration parameters. See the "Diagnostics Pane: Data Validity" reference page in *Simulink Graphical User Interface* for a description of the data validity settings that apply to parameters.

See "Run-Time Parameters" on page 8-8 for more information on run-time parameters.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime3.c` used in `sfcndemo_runtime.mdl`.

See Also

`ssRegAllTunableParamsAsRunTimeParams`

ssRegAllTunableParamsAsRunTimeParams

Purpose

Register all tunable parameters as run-time parameters

Syntax

```
void ssRegAllTunableParamsAsRunTimeParams(SimStruct *S,
  const char_T *names[]))
```

Arguments

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

`names`
: Array of names for the run-time parameters.

Description

Use this function in `mdlSetWorkWidths` to register all tunable dialog parameters as run-time parameters. Specify the names of the run-time versions of the parameters in the `names` array.

Note The Simulink engine 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`. See "Run-Time Parameters" on page 8-8 for more information.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_cplx.c` used in `sfcndemo_cplx.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_matadd.c` used in `sfcndemo_matadd.mdl`.

See Also

`mdlSetWorkWidths`, `ssSetNumRunTimeParams`, `ssSetRunTimeParamInfo`

Purpose

Register the method to handle dimensions mode propagation for each input port.

Syntax

```
void ssRegMdlSetInputPortDimensionsModeFcn(SimStruct *S, mdlSetInputPo
```

Arguments

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

`fcn`
: Function pointer corresponding to the function being registered.

Returns

No return value.

Description

Use this function in mdlInitializeSizes to register the input port dimensions mode propagation method.

Languages

C, C++

Example

See the S-function
`*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_hold`

See Also

`mdlInitializeSizes`, `mdlSetInputPortDimensionsModeFcn`

ssRegisterDataType

Purpose

Register a custom data type

Syntax

```
DTypeId ssRegisterDataType(SimStruct *S, char *name)
```

Arguments

`S`
  SimStruct representing an S-Function block.

`name`
  Name of the custom data type.

Returns

The `DTypeId` associated with the registered data type. Otherwise, reports an error and returns `INVALID_DTYPE_ID`.

Description

Register a custom data type. Each data type must be a valid MATLAB identifier. That is, the first character 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`.

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Note You can call this function to get the data type ID associated with a registered data type. For a list of built-in values for the data type ID, see `ssGetInputPortDataType`.

Languages

C, C++

Example

The following example registers a custom data type named `Color`.

```
DTypeId id = ssRegisterDataType(S, "Color");
if(id == INVALID_DTYPE_ID) return;
```

See “Custom Data Types” on page 8-29 for a more detailed example showing how to use this function.

See Also

`ssRegisterDataType`

ssRegisterTypeFromNamedObject

Purpose

Register a custom data type from a `Simulink.AliasType`, `Simulink.StructType`, or `Simulink.NumericType` object.

Syntax

```
void ssRegisterTypeFromNamedObject(SimStruct *S, char *name, int* id)
```

Arguments

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

`name`
: Name of the Simulink object to assign to the custom data type.

`id`
: An integer whose value is the numeric data type identifier after the call to `ssRegisterTypeFromNamedObject`.

Description

Use in `mdlInitializeSizes`, to register a custom data type from a `Simulink.AliasType`, `Simulink.StructType`, or `Simulink.Numeric` object named `name`.

If the registration was successful, you can declare S-function parameters, DWork vectors, or input and output ports to be of this data type, using the corresponding numeric data type identifier `id`. If the registration was not successful, `id` is set to `INVALID_DTYPE_ID`.

Note You may not register a custom data type from a `Simulink.Bus` object or `Simulink.Numeric` object with unspecified scaling.

Languages

C, C++

Example

The following example registers a custom data type from the `Simulink.Numeric` type named `mydouble`. It then specifies that a DWork vector and the first output port be of this data type.

```
int dtype;
char *name = "mydouble";
ssRegisterTypeFromNamedObject(S, name, &dtype);
```

```
ssSetDWorkDataType(S, 0, dtype);
ssSetOutputPortDataType(S, 0, dtype);
```

See Also `ssSetDataTypeSize`

ssRTWGenIsAccelerator

Purpose

Determine if the model is running in Accelerator mode.

Syntax

```
boolean_T ssRTWGenIsAccelerator(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Returns

The Boolean value `true` if the model has compiled or is compiling for Accelerator mode simulation. Returns `false` otherwise.

Description

Use this macro in an S-function to determine whether the model is currently simulating in Accelerator mode.

Languages

C, C++

See Also

`ssRTWGenIsModelReferenceRTWTarget`, `ssRTWGenIsModelReferenceSimTarget`, `ssRTWGenIsCodeGen`, `ssIsExternal`, `ssGetSimMode`

Purpose

Identify any code generation that is not used by the Accelerator.

Syntax

```
boolean_T ssRTWGenIsCodeGen(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Returns

The Boolean value `true` if the model is generating code for any purpose other than Accelerator mode simulation. Returns `false` if the model is not generating code or is generating code for Accelerator mode simulation.

Description

Use this macro in an S-function to determine whether the model is generating code.

Languages

C, C++

See Also

`ssRTWGenIsModelReferenceRTWTarget`, `ssRTWGenIsModelReferenceSimTarget`, `ssRTWGenIsAccelerator`, `ssIsExternal`, `ssGetSimMode`

ssRTWGenIsModelReferenceRTWTarget

Purpose

Determine if the model reference Real-Time Workshop target is generating

Syntax

```
boolean_T ssRTWGenIsModelReferenceRTWTarget(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Returns

The Boolean value `true` if the model reference Real-Time Workshop target is building. Returns `false` if the model reference simulation target is building.

Description

Use this macro in an S-function that can be contained in a referenced submodel (See "Referencing Models" in *Using Simulink*) to determines whether the submodel containing this S-function is currently generating a simulation target or Real-Time Workshop target.

Languages

C, C++

See Also

`ssRTWGenIsModelReferenceSimTarget`

Purpose

Determine if the model reference simulation target is generating

Syntax

```
boolean_T ssRTWGenIsModelReferenceSimTarget(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Returns

The Boolean value `true` if the model reference simulation target is building. Returns `false` if the model reference Real-Time Workshop target is building.

Description

Use this macro in an S-function that can be contained in a referenced submodel (See “Referencing Models” in *Using Simulink*) to determines whether the submodel containing this S-function is currently generating a simulation target or Real-Time Workshop target.

Languages

C, C++

See Also

`ssRTWGenIsModelReferenceRTWTarget`

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.

Returns

The Boolean value `true` if both `st` and `ot` are equal to `INHERITED_SAMPLE_TIME`. Otherwise, returns `false`.

Description

The Simulink engine 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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_port_triggered.` used in `sfcndemo_port_triggered.mdl`.

See Also

`ssSampleAndOffsetAreTriggeredOrAsync`

Purpose

Determine whether a sample time and offset value pair indicate a triggered or asynchronous sample time

Syntax

```
boolean_T ssSampleAndOffsetAreTriggeredOrAsync(real_T st, real_T ot)
```

Arguments

`st`
: The sample time.

`ot`
: The offset time

Returns

The Boolean value `true` if `st` is equal to `INHERITED_SAMPLE_TIME` (`-1`) and `ot` is either `INHERITED_SAMPLE_TIME` (`-1`) or any other negative integer.

Description

The Simulink engine 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. The Simulink engine sets the offset time to an integer value less than `-1` when the block resides in an asynchronous function-call subsystem. By invoking this macro on its sample time/offset pairs, an S-function can determine whether it resides in a triggered subsystem or an asynchronous function-call subsystem.

Languages

C, C++

See Also

`ssSampleAndOffsetAreTriggered`

ssSetBlockReduction

Purpose

Request that the Simulink engine attempt to reduce a block

Syntax

```
uint_T ssSetBlockReduction(SimStruct *S, unsigned int_T flag)
```

Arguments

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

`flag`
: If not zero, the Simulink engine should attempt to reduce this block.

Returns

`0` if `flag` is `0` and `1`, otherwise.

Description

Use this macro to ask the engine to reduce this block. A block is reducible if it can be eliminated from the model without affecting the model's behavior. The engine optimizes performance by skipping execution of reducible blocks during model simulation. In particular, the engine does not invoke the `mdlStart`, `mdlUpdate`, and `mdlOutputs` methods of reducible blocks. Further, the engine executes the `mdlTerminate` method of a reduced block only if the block has set the `SS_OPTION_CALL_TERMINATE_ON_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.
- A block must have the same number of outputs as inputs or no outputs.
- A block cannot have inputs that are bus signals.
- A block cannot have continuous state.
- A block cannot have discrete states while the model is logging states.
- A block cannot have zero crossings.
- A block cannot have tunable parameters.

If a block fails to meet any of these criteria, the engine includes the block in the simulation regardless of whether the block has requested reduction.

See the “Block reduction” reference page in *Simulink Graphical User Interface* for further details. Note, if you want to enable dead branch elimination, do not request block reduction. Instead, set the `SS_OPTION_NONVOLATILE` option using `ssSetOptions`.

Your S-function must invoke this macro before the engine would otherwise invoke the S-function’s `mdlStart` method (see the callback flow diagram in “How the Simulink Engine Interacts with C S-Functions” on page 4-77). 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, C++

See Also

`ssGetBlockReduction`

ssSetCallSystemOutput

Purpose

Specify that an output port is issuing a function call

Syntax

```
void ssSetCallSystemOutput(SimStruct *S, int_T element_index)
```

Arguments

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

`element_index`
: Index of the element of the first output port that is issuing a function call.

Description

Use in `mdlInitializeSampleTimes` to specify that the output port element specified by ***`element_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)`. See “Function-Call Subsystems” on page 8-60 for more information.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_fcncall.c`.

See Also

`ssCallSystemWithTid`

Purpose

Sets the current size corresponding to dimension dIdx of the output signal at port pIdx.

Syntax

```
void ssSetCurrentOutputPortDimensions(SimStruct *S,int_T pIdx, int_T d
```

Arguments

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

`pldx`
: Output port index being set.

`dIdx`
: Index of dimension being set.

`val`
: Current size value to set for dimension dIdx.

Returns

No return value.

Languages

C, C++

Example

See the S-function
`matlabroot``/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_cond`

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 the data type.

`size`
: Size of the custom data type in bytes.

Returns `1` (`true`) if successful. Otherwise, returns `0` (`false`).

Description Sets the size of the data type specified by `id` to `size`. Use this macro in `mdlInitializeSizes` to set the size of a data type you have registered. See “Custom Data Types” on page 8-29 for more information on registering custom data types.

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages C, C++

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

status = ssSetDataTypeSize(S, id, 4);
if(status == 0) return;
```

See Also `ssRegisterDataType`, `ssGetDataTypeSize`

ssSetDataTypeZero

Purpose

Set the 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 the data type.

`zero`
: Zero representation of the data type specified by `id`.

Returns

`1` (`true`) if successful. Otherwise, returns `0` (`false`) and reports an error.

Description

Successfully sets the zero representation of the data type specified by `id` to `zero` 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 fails and reports an error. Because this macro reports any error that occurs, you do not need to use `ssSetErrorStatus` to report the error. See "Custom Data Types" on page 8-29 for more information on registering custom data types.

Note This macro makes a copy of the zero representation of the data type for the Simulink engine to use. Thus, your S-function does not have to maintain the original in memory.

The Real-Time Workshop product does not support S-functions that contain custom data types. Attempting to generate code for a model that contains this macro results in an error.

Languages

C, 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

```
CSignal_T ssSetDWorkComplexSignal(SimStruct *S, int_T vector,
  CSignal_T csig)
```

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)-1`.

`numType`
: Numeric type, either `COMPLEX_YES` or `COMPLEX_NO`.

Returns

`1` (`COMPLEX_YES`), `0` (`COMPLEX_NO`), or `-1` (`COMPLEX_INHERITED`) depending on the value specified by `csig`.

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). For more information on using DWork vectors, see Chapter 7, "Using Work Vectors".

Languages

C, C++

Examples

For more information on using DWork vectors, see Chapter 7, "Using Work Vectors".

See Also

`ssSetDWorkDataType`, `ssGetNumDWork`

Purpose

Specify the data type of a data type work vector

Syntax

```
DTypeId 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)-1`.

`dtID`
: ID of a data type.

Returns

The data type ID specified by `dtID`. Returns `-1` if `dtID` is `DYNAMICALLY_TYPED`.

Description

Use in `mdlInitializeSizes` or `mdlSetWorkWidths` to set the data type of the specified work vector. For a list of built-in values for the data type ID `dtId`, see `ssGetInputPortDataType`. For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl`.

See Also

`ssSetDWorkWidth`, `ssGetNumDWork`

ssSetDWorkName

Purpose

Specify the name of a data type work vector

Syntax

```
const char_T *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)-1`.

`name`
: Name of a work vector.

Returns

The name of the DWork vector entered in `name`.

Description

Use in `mdlInitializeSizes` or in `mdlSetWorkWidths` to specify a name for the specified data type work vector. The Real-Time Workshop product uses this name to label the work vector in generated code. If you do not specify a name, the Real-Time Workshop product generates a name for the DWork vector in the code. For more information on using DWork vectors, see Chapter 7, "Using Work Vectors".

Note `ssSetDWorkName` stores only the pointer to the name string. Therefore, the name string must be in persistent memory; it cannot be a local variable.

Languages

C, C++

Example

The following example dynamically generates and sets the names of multiple DWork vectors.

```
#define MDL_SET_WORK_WIDTHS
#if defined(MDL_SET_WORK_WIDTHS)
static void mdlSetWorkWidths(SimStruct *S)
```

```
{
int_T i;

/* Number of DWork vectors based on input width */
ssSetNumDWork(S, ssGetInputPortWidth(S,0));

/* Malloc space to store the names of the DWork vectors */
name=(char_T *)malloc(sizeof(char_T)*ssGetNumDWork(S)*16);

for (i = 0; i < ssGetNumDWork(S); i++) {
   sprintf(&name[i*16], "DWork%d", i+1);
   ssSetDWorkName(S, i, &name[i*16]);
   }
}
#endif /* MDL_SET_WORK_WIDTHS */
```

See the S-functions in `sfcndemo_sfun_rtwdwork.mdl` for a complete example using DWork vectors.

See Also

`ssGetDWorkName`, `ssSetNumDWork`

ssSetDWorkRequireResetForSignalSize

Purpose

Set the block flag for resetting the dIndex Dwork size upon subsystem reset.

Syntax

```
void ssSetDWorkRequireResetForSignalSize (SimStruct *S, int dIndex, SS_ V
```

Arguments

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

`dIndex`
: Dwork index.

`type`
: Enum value corresponding to the signal size compute type.

Returns

No return value.

Description

Use this function in mdlSetWorkWidths to configure the need of resetting the size of the Dworks indexed by dIndex in case there is a subsystem reset. The possible types are SS_VARIABLE_SIZE_STATE_NO_NEED_RESET SS_VARIABLE_SIZE_REQUIRE_STATE_RESET Use SS_VARIABLE_SIZE_REQUIRE_STATE_RESET when the state requires reset when input size changes, and SS_VARIABLE_SIZE_STATE_NO_NEED_RESET otherwise.

Languages

C, C++

Example

See the S-function
`matlabroot``/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_holdSta`

See Also

`mdlSetWorkWidths`

Purpose

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

Syntax

```
char_T *ssSetDWorkRTWIdentifier(SimStruct *S, int_T vector, char_T *ic
```

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)-1`.

`id`

DWork vector Identifier.

Returns

A pointer (char_T *) to the Real-Time Workshop identifier entered in `id`.

Description

Specifies `id` as the identifier used in code generated by the Real-Time Workshop product to declare the DWork vector specified by `vector`. For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

This function must only be called from the `mdlInitializeSizes` or `mdlSetWorkWidths` functions.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl`.

See Also

`mdlInitializeSizes`

ssSetDWorkRTWIdentifierMustResolveToSignalObject

Purpose

Specify if a DWork vector resolves to a `Simulink.Signal` object

Syntax

```
unsigned int_T ssSetDWorkRTWIdentifierMustResolveToSignalObject(SimStruct
 int_T vector, unsigned int_T flag)
```

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)-1`.

`flag`
: Flag to control if the DWork vector resolves to a `Simulink.Signal` object, either `0`, `1`, or `2`.

Returns

The value for `flag` if flag is `0`, `1`, or `2`. Otherwise, returns `0`.

Description

Use this function in `mdlInitializeSizes` to set a flag that controls if the DWork vector specified by `vector` resolves to a `Simulink.Signal` object. The input argument `flag` takes one of the following three values.

- `0` instructs the Simulink engine to try to resolve the DWork vector to a `Simulink.Signal` object. The engine only tries to resolve the DWork vector to a `Simulink.Signal` object if implicit signal resolution is enabled. The Data Validity parameter **Signal resolution** on the Diagnostics pane of the Configuration Parameters dialog box controls implicit signal resolution. When this option is set to `Explicit only`, the engine interprets a flag of `0` as it would a flag of `2`. See the “Signal resolution” reference page in *Simulink Graphical User Interface* for more information on implicit signal resolution.
- `1` declares that the DWork vector must resolve to a `Simulink.Signal` object. The engine invokes an error if it cannot resolve the DWork vector to a `Simulink.Signal` object.
- `2` instructs the engine to not try to resolve the DWork vector to a `Simulink.Signal` object.

For more information on using DWork vectors, see Chapter 7, "Using Work Vectors".

Languages C, C++

See Also `ssGetDWorkRTWIdentifierMustResolveToSignalObject`

ssSetDWorkRTWStorageClass

Purpose

Specify the storage class of a DWork vector in code generated from the associated S-function

Syntax

```
ssRTWStorageType ssSetDWorkRTWStorageClass(SimStruct *S, int_T vector,
 ssRTWStorageType sc)
```

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)-1`.

`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
```

Returns

The `ssRTWStorageType` value entered as `sc`. Invokes an error if `sc` is not a valid storage class.

Description

Sets `sc` as the storage class of the DWork vector specified by `vector`. The storage class is a code-generation attribute that determines how code generated by the Real-Time Workshop product for this S-function allocates memory for this work vector (see “Signal Storage Concepts” in the *Real-Time Workshop User’s Guide*). For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl`.

See Also

`ssGetDWorkRTWStorageClass`

ssSetDWorkRTWTypeQualifier

Purpose

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

Syntax

```
char_T *ssSetDWorkRTWTypeQualifier(SimStruct *S, int_T vector,
 char_T *tq)
```

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)-1`.

`tq`
: Type qualifier.

Returns

The C type qualifier entered in `tq`.

Description

Sets `tq` as the C type qualifier (e.g., `const`) used to declare the DWork vector specified by `vector` in code generated by the Real-Time Workshop product from the associated S-function. For more information on using DWork vectors, see Chapter 7, "Using Work Vectors".

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl`.

See Also

`ssGetDWorkRTWTypeQualifier`

Purpose

Specify how DWork vector is used in S-function

Syntax

```
ssDWorkUsageType ssSetDWorkUsageType(SimStruct *S, int_T vector,
 ssDWorkUsageType type)
```

Arguments

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

`vector`
: Index of the data type work vector.

`type`
: Usage type of the DWork vector.

Returns

The usage type entered in `type`.

Description

Use this function in `mdlInitializeSizes` to set the usage type for the DWork vector specified by `vector`. Permissible values are:

- `SS_DWORK_USED_AS_DWORK`

  The DWork vector is used as a data type work vector. This is the default value.

- `SS_DWORK_USED_AS_DSTATE`

  The DWork vector is used to store the block's discrete states.

- `SS_DWORK_USED_AS_SCRATCH`

  The DWork vector is used as a temporary scratch work vector.

- `SS_DWORK_USED_AS_MODE`

  The DWork vector is used as a mode vector.

For more information on using DWork vectors, see Chapter 7, "Using Work Vectors".

Languages

C, C++

Examples

For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

See Also

`ssGetDWorkUsageType`

Purpose

Specify that a data type work vector is used as a discrete state vector

Syntax

```
int_T 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)-1`.

`usage`
: How this vector is used. A value of `1` indicates that the work vector is to be used to store the block's discrete states (`SS_DWORK_USED_AS_DSTATE`), a value of `0` indicates that the work vector is to be used as a work vector (`SS_DWORK_USED_AS_DWORK`).

Returns

`0` if usage is `SS_DWORK_USED_AS_DWORK` (`0`), otherwise returns `1`.

Description

Use in `mdlInitializeSizes` or `mdlSetWorkWidths` to specify if the DWork vector `vector` is used to store the block's discrete states, `SS_DWORK_USED_AS_DSTATE` (`1`), or not, `SS_DWORK_USED_AS_DWORK` (`0`), 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 the Simulink engine to log the discrete states to the workspace at the end of a simulation, if the user has selected the **Save to Workspace** options on the **Data Import/Export** pane of the Configuration Parameters dialog box.

Languages

C, C++

Examples

For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

See Also

`ssGetDWorkUsedAsDState`

Purpose

Specify the width of a data type work vector

Syntax

```
int_T 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)-1`.

`width`
: Number of elements in the work vector.

Returns

The number of elements passed in through `width`.

Description

Use in `mdlInitializeSizes` or in `mdlSetWorkWidths` to set the number of elements in the specified data type work vector. For more information on using DWork vectors, see Chapter 7, “Using Work Vectors”.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl`.

See Also

`ssGetDWorkWidth`, `ssSetDWorkDataType`, `ssSetNumDWork`

ssSetErrorStatus

Purpose

Report an error

Syntax

`void ssSetErrorStatus(SimStruct *S, const char_T *msg)`

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. If you use `sprintf` to format the error message, you must allocate memory for the message. For example:

```
static char *msg[35];
sprintf(msg,"Expected number of parameters: %d",ssGetNumSFcnParams(S));
ssSetErrorStatus(S,msg);
```

This function causes the Simulink engine to stop and display the specified error message. The function does not generate an exception. Thus you can use it in your S-function to avoid creating exceptions when reporting errors.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl`.

See Also `ssWarning`

ssSetExplicitFCSSCtrl

Purpose

Specify whether this S-function explicitly enables and disables the function-call subsystem that it calls

Syntax

```
void ssSetExplicitFCSSCtrl(SimStruct *S, unsigned int_T explicit)
```

Arguments

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

`explicit`
: `1` if this S-function explicitly enables and disables the function-call subsystems it enables

Description

Specify `1` as the value of `explicit` if `S` explicitly enables or disables the function-call subsystems that it calls. Call this macro in either `mdlInitializeSizes` or `mdlInitializeSampleTimes`. Use `ssEnableSystemWithTid` and `ssDisableSystemWithTid` to subsequently enable and disable the function-call subsystem. See "Function-Call Subsystems" on page 8-60 for more information on interfacing S-functions and function-call subsystems.

Note If the S-function is set to explicitly enable and disable a function-call subsystem, the S-function must enable the function-call subsystem using `ssEnableSystemWithTid` prior to calling it using `ssCallSystemWithTid`.

Languages

C, C++

See Also

`ssGetExplicitFCSSCtrl`, `ssEnableSystemWithTid`, `ssDisableSystemWithTid`

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`. This macro is for internal use. User-written S-functions should not use the `ssSetExternalModeFcn` macro.

Languages

C, C++

See Also

`ssCallExternalModeFcn`

ssSetInputPortComplexSignal

Purpose

Set the numeric type (real or complex) of an input port

Syntax

```
CSignal_T ssSetInputPortComplexSignal(SimStruct *S, int_T port,
 CSignal_T csig)
```

Arguments

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

`port`
: Index of an input port.

`csig`
: 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).

Returns

`1` (`COMPLEX_YES`), `0` (`COMPLEX_NO`), or `-1` (`COMPLEX_INHERITED`) depending on the value specified by `csig`.

Description

Use this function in `mdlInitializeSizes` to initialize the input port numeric 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`. In this case, the S-function must provide `mdlSetInputPortComplexSignal` and `mdlSetDefaultPortComplexSignals` methods to enable the numeric type to be set correctly during signal propagation. The default numeric type of an input port is `real`.

Languages

C, 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 the S-function *matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_cplx.c used in sfcndemo_cplx.mdl for a complete example that uses this function.

See Also ssGetInputPortComplexSignal

ssSetInputPortDataType

Purpose

Set the data type of an input port

Syntax

```
DTypeId ssSetInputPortDataType(SimStruct *S, int_T port, DTypeId id)
```

Arguments

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

`port`
: Index of an input port.

`id`
: ID of the data type accepted by `port`.

Returns

The data type ID specified by `id`. Returns `-1` if `id` is `DYNAMICALLY_TYPED`.

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`. In this case, the S-function must provide `mdlSetInputPortDataType` and `mdlSetDefaultPortDataTypes` methods to enable the data type to be set correctly during signal propagation.

For a list of the built-in data types, see `BuiltInDTypeID` in *`matlabroot`*`/simulink/include/simstruc_types.h`.

Note The data type of an input port is `double` (`real_T`) by default.

Languages

C, 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 the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl` and the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl` for complete examples that use this function.

See Also

`ssGetInputPortDataType`

Purpose

Specify information about the dimensionality of an input port

Syntax

```
int_T 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 DYNAMICALLY_SIZED]`.
- `width` equals the total number of elements in the signal, e.g., 12 for a 3-by-4 matrix signal or 8 for an 8-element vector signal, or

DYNAMICALLY_SIZED if the total number of elements is determined dynamically.

Note Use the macro, DECL_AND_INIT_DIMSINFO, to declare and initialize an instance of this structure.

Returns

1 if successful; otherwise, 0.

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. In this case, the S-function must provide mdlSetInputPortDimensionInfo and mdlSetDefaultPortDimensionInfo methods to enable the signal dimensions to be set correctly during signal propagation.

Languages

C, C++

Example

The following example specifies that input port 0 accepts 2-by-2 matrix signals.

```
{
    DECL_AND_INIT_DIMSINFO(di);
    int_T dims[2];

    di.numDims = 2;
    dims[0] = 2;
    dims[1] = 2;
    di.dims = dims;
    di.width = 4;
    ssSetInputPortDimensionInfo(S, 0, &di);
}
```

See the S-function *matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_matadd.c used in sfcndemo_matadd.mdl and the S-function *matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c used in sfcndemo_sdotproduct.mdl for complete examples that use this function.

See Also

ssSetInputPortMatrixDimensions, ssSetInputPortVectorDimension

Purpose Sets the dimensions mode of the input port indexed by pIdx

Syntax

```
Void ssSetInputPortDimensionsMode(SimStruct *S,int_T pIdx, DimensionsM
```

Arguments

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

`pIdx`
: Input port index being polled.

`mode`
: Enum value corresponding to the ports dimensions mode.

Returns No return value

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_cond`

ssSetInputPortDimsSameAsOutputPortDims

Purpose

Set the dimensions of ouput port outIdx to be equal than the dimensions of input port inpIdx. This method is called from mdlSetWorkWidths.

Syntax

```
void ssSetInputPortDimsSameAsOutputPortDims(SimStruct *S, int_T inpIdx, i
```

Arguments

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

`inpIdx`
: Input port index.

`outIdx`
: Output port index.

Returns

No return value

Description

Use this function in mdlSetWorkWidths to set the dimensions of output port outIdx to be equal to the dimensions of input port inpIdx without the need of using ssSetCurrentOutputPortDimensions for each output dimension. When this function is issued, it indicates that the S-Function does not set the outport port current dimension.

Languages

C, C++

Purpose

Specify the direct feedthrough status of a block's ports

Syntax

```
void ssSetInputPortDirectFeedThrough(SimStruct *S, int_T port,
 int_T dirFeed)
```

Arguments

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

`port`
: Index of the input port whose direct feedthrough property is being set.

`dirFeed`
: Direct feedthrough status of the block specified by `port`.

Description

Use in `mdlInitializeSizes` or `mdlSetWorkWidths` (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.

Note The `ssSetInputPortDirectFeedThrough` macro becomes a function when you compile your S-function in debug mode (`mex -g`).

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl` and the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl`.

See Also `ssGetInputPortDirectFeedThrough`

Purpose

Specify whether a port accepts signal frames

Syntax

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

Arguments

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

`port`
: Index of an input port.

`frameData`
: Type of signal accepted by `port`. Acceptable values are `FRAME_INHERITED` (either frame or unframed input), `FRAME_NO` (unframed input only), and `FRAME_YES` (framed input only).

Description

Use in `mdlInitializeSizes` to specify whether the port accepts frame signals. By default, input ports do not accept frame signals (`FRAME_NO`). If the S-function specifies `FRAME_INHERITED` for any of its ports, the S-function should include an `mdlSetInputPortFrameData` callback method. This callback method sets the frame status of ports that inherit their frame status to the value that the Simulink engine assigns to them during frame status propagation. The engine passes the assigned status to the callback as an argument. The callback method can also use this function to assign the frame status of other ports on the block whose frame status depends on the frame status of the port that inherits its frame status.

The use of frame-based signals (`frameData` has a value of `FRAME_YES`) requires a Signal Processing Blockset product license.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl`.

See Also `ssGetInputPortFrameData`, `mdlSetInputPortFrameData`

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`.

Returns

`1` if successful; otherwise, `0`.

Description

Use this function to specify that `port` accepts an `m`-by-`n` matrix signal. If either dimension is `DYNAMICALLY_SIZED`, the other must be `DYNAMICALLY_SIZED` or `1`. If either dimension is dynamically sized, the S-function must provide `mdlSetInputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` methods to enable the signal dimensions to be set correctly during signal propagation.

Languages

C, C++

Example

The following example specifies that input port 0 accepts 2-by-2 matrix signals.

```
ssSetInputPortMatrixDimensions(S,  0, 2, 2);
```

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_frmunbuff.c` used in `sfcndemo_frame.mdl` for a complete example that uses this function. Running this model requires a Signal Processing Blockset license.

Purpose

Specify the offset time of an input port

Syntax

```
real_T ssSetInputPortOffsetTime(SimStruct *S,
  int_T inputPortIdx, real_T offset)
```

Arguments

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

`inputPortIdx`
: Index of the input port whose offset time is being set.

`offset`
: Offset time.

Returns

The `real_T` value of the offset time passed into the macro.

Description

Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the sample time offset for each input port index. Input port index numbers start at 0 and end at the total number of input ports minus 1. You can use this macro in conjunction with `ssSetInputPortSampleTime` if you have specified port-based sample times for your S-function.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also

`ssSetNumInputPorts`, `ssSetInputPortSampleTime`

ssSetInputPortOptimOpts

Purpose

Specify reusability of the memory allocated to the input port of an S-function

Syntax

```
void ssSetInputPortOptimOpts(SimStruct *S, int_T port, uint_T val)
```

Arguments

`S`

SimStruct representing an S-Function block.

`port`

Index of an input port of `S`.

`val`

Reusability of `port`. Permissible values are

- `SS_NOT_REUSABLE_AND_GLOBAL`
- `SS_REUSABLE_AND_LOCAL`
- `SS_REUSABLE_AND_GLOBAL`
- `SS_NOT_REUSABLE_AND_LOCAL`

Description

Use this macro to specify the reusability and scope of the memory allocated to an S-function input port. The reusability indicates if the memory associated with the input port can be overwritten, or not. You must indicate an input port is reusable if you use the `ssSetInputPortOverWritable` macro to specify the input port's memory can be overwritten by one of the output ports.

The Simulink engine disregards the memory scope setting, instead treating all S-function ports as global during simulation.

Note The Real-Time Workshop product uses the memory scope setting you specified when generating code from a model. The Real-Time Workshop product attempts to declare local variables for any inputs with a local scope. If your S-function uses the inputs in a way that precludes using a local scope, the generated code uses global variables for the inputs. See "Writing S-Functions for Multirate Multitasking Environments" and "Writing S-Functions That Specify Port Scope and Reusability" in the *Real-Time Workshop User's Guide* for more information.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl` and the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl`.

See Also

`ssGetInputPortOptimOpts`, `ssSetOutputPortOptimOpts`, `ssSetInputPortOverWritable`

Purpose

Specify whether one of an S-function's input ports can be overwritten by one of its output ports

Syntax

```
void ssSetInputPortOverWritable(SimStruct *S, int_T port, int_T
 isOverwritable)
```

Arguments

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

`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 `port` is overwritable by one of the S-function's output ports. The Simulink engine uses this setting as one criterion in determining whether one of the output ports of this S-function can share memory with `port`. If `isOverwritable=1` and the other criteria are satisfied, the engine allocates a common block of memory for the input port and one of the S-function's output ports, thus reducing simulation memory requirements. The default is `isOverwritable=0`, which means that `port` cannot share memory with any of the S-function's output ports.

Note If you set an input port to be overwritable, you must also specify that the input port and at least one of the S-function's output ports are reusable. Use `ssSetInputPortOptimOpts` and `ssSetOutputPortOptimOpts` to do this.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl` and the S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl`.

See Also

`ssSetNumInputPorts`, `ssSetInputPortOptimOpts`, `ssSetOutputPortOptimOpts`, `ssGetInputPortBufferDstPort`

ssSetInputPortRequiredContiguous

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.

`port`
: Index of an input port.

`flag`
: True (1) 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 (0). Hence, the default method for accessing the input signals is `ssGetInputSignalPtrs`.

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_io_handling2.c` used in `sfcndemo_io_handling.mdl`.

See Also `mdlInitializeSizes`, `mdlSetWorkWidths`, `ssGetInputPortSignal`, `ssGetInputPortSignalPtrs`

Purpose

Specify the sample time of an input port

Syntax

```
real_T ssSetInputPortSampleTime(SimStruct *S, int_T inputPortIdx,
 real_T period)
```

Arguments

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

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

`period`
: Sample period.

Returns

The `real_T` value of the sample time passed into the macro.

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. For a continuous sample time, specify `period` as `CONTINUOUS_SAMPLE_TIME`. To inherit the sample time, specify period as `INHERITED_SAMPLE_TIME`. You should use this macro only if you have specified port-based sample times.

If the S-function specifies `INHERITED_SAMPLE_TIME` for any of its ports, the S-function should include an `mdlSetInputPortSampleTime` callback method. The callback method should set the sample time and offset of ports that inherit their sample time to the status that the Simulink engine assigns to them using its sample time propagation rules. The callback method can also assign the sample times and offsets of other ports on the block whose sample times are inherited.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also `ssSetNumInputPorts`, `ssSetInputPortOffsetTime`

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 the vector or `DYNAMICALLY_SIZED`.

Returns

`1` if successful; otherwise, `0`.

Description

Specifies that `port` accepts a `w`-element vector signal. If the width is dynamically sized, the S-function must provide `mdlSetInputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` methods to enable the signal dimensions to be set correctly during signal propagation.

Note This function and the `ssSetInputPortWidth` macro are functionally identical.

Languages

C, C++

Example

The following example specifies that input port 0 accepts an 8-element matrix signal.

```
ssSetInputPortVectorDimension(S, 0, 8);
```

See Also

`ssSetInputPortWidth`

ssSetInputPortWidth

Purpose

Specify the width of an input port

Syntax

```
int_T ssSetInputPortWidth(SimStruct *S, int_T port, int_T width)
```

Arguments

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

`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. If the width is dynamically sized, the S-function must provide `mdlSetInputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` methods to enable the signal dimensions to be set correctly during signal propagation.

Note This macro and `ssSetInputPortVectorDimension` are functionally identical with the exception of their returns. The`ssSetInputPortVectorDimension` returns 1 if successful, and 0 otherwise.

Languages

C, C++

Example

See the S-function `sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl`.

See Also

`ssSetNumInputPorts`, `ssSetOutputPortWidth`, `ssSetInputPortVectorDimension`

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.

Returns

The `int_T` value passed into the macro.

Description

Sets the `idx` element of the S-function'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.

Languages

C, C++

Example

The following statement

```
ssSetIWorkValue(S, 0, 1);
```

sets the first element of the work vector to `1`.

See Also

`ssGetNumIWork`, `ssGetIWork`, `ssGetIWorkValue`

ssSetModelReferenceNormalModeSupport

Purpose

Specify if S-function can be used in referenced model simulating in normal mode

Syntax

```
void ssSetModelReferenceNormalModeSupport(SimStruct *S,
 ssModelReferenceNormalModeSupport mode)
```

Arguments

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

`mode`
: Flag for normal mode simulation support when the S-Function block is used in a referenced model. Options are `DEFAULT_SUPPORT_FOR_NORMAL_MODE` (0) or `MDL_START_AND_MDL_PROCESS_PARAMS_OK` (1).

Description

Use in `mdlInitializeSizes` to specify if an S-function with both an `mdlStart` and an `mdlProcessParameters` method can be used in a referenced model simulating in normal mode. Permissible values are:

- `DEFAULT_SUPPORT_FOR_NORMAL_MODE`: The Simulink engine produces an error if the S-function is in a referenced model simulating in normal mode.
- `MDL_START_AND_MDL_PROCESS_PARAMS_OK`: The Simulink engine allows normal mode simulation of the S-function in a referenced model.

The `ssSetModelReferenceNormalModeSupport` flag indicates to the engine if the code in `mdlProcessParameters` is independent of the code in `mdlStart`. This information is important because the engine modifies its normal mode simulation process for S-functions in a referenced model. If the S-function is not in a referenced model, the engine always executes the `mdlStart` method prior to the `mdlProcessParameters` method. However, during normal mode simulation of referenced models, the engine may decide to execute the S-function's `mdlProcessParameters` method prior to `mdlStart`. If the mdlProcessParameters method requires data initialized in mdlStart, the engine cannot successfully change the order of execution.

By default, the engine produces an error if it finds an S-function with both an mdlStart and an mdlProcessParameters method in a referenced model simulating in normal mode. The default behavior is equivalent to specifying the DEFAULT_SUPPORT_FOR_NORMAL_MODE option for ssSetModelReferenceNormalModeSupport. If the S-function does not depend on the execution order of these two methods, specify the MDL_START_AND_MDL_PROCESS_PARAMS_OK option to enable normal mode simulation.

Languages

C, C++

Example

See the S-function *matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_frmad.c used in sfcndemo_frame.mdl for an example that uses this function. Running this model requires a Signal Processing Blockset license.

ssSetModelReferenceSampleTimeDefaultInheritance

Purpose

Specify that a submodel containing this S-function can inherit its sample time from its parent model

Syntax

```
void ssSetModelReferenceSampleTimeDefaultInheritance(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Description

Use this macro in any callback from `mdlInitializeSizes` to `mdlSetWorkWidths` to specify that submodels containing this S-function can inherit their sample times from their parent model. You should only use this macro if your S-function inherits its sample time, but its output does not depend on the value of the inherited sample time. If the S-function output does depend on the inherited sample time, use `ssSetModelReferenceSampleTimeDisallowInheritance`.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl`.

See Also

`ssSetModelReferenceSampleTimeDisallowInheritance`, `ssSetModelReferenceSampleTimeInheritanceRule`

ssSetModelReferenceSampleTimeDisallowInheritance

Purpose

Specify that the use of this S-function in a submodel prevents the submodel from inheriting its sample time from its parent model

Syntax

```
void ssSetModelReferenceSampleTimeDisallowInheritance(SimStruct *S)
```

Arguments

`S`

SimStruct representing an S-Function block.

Description

Use this macro in any callback from `mdlInitializeSizes` to `mdlSetWorkWidths` to specify that submodels containing this S-function cannot inherit their sample times from their parent model. You should only use this macro if your S-function inherits its sample time and its output depends on the value of the inherited sample time. See "Specifying Model Reference Sample Time Inheritance" on page 8-48 for more information.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_frmad.c` used in `sfcndemo_frame.mdl` for an example that uses this function. Running this model requires a Signal Processing Blockset license.

See Also

`ssSetModelReferenceSampleTimeDefaultInheritance`, `ssSetModelReferenceSampleTimeInheritanceRule`

ssSetModelReferenceSampleTimeInheritanceRule

Purpose

Specify whether use of this S-function in a submodel prevents the submodel from inheriting its sample time from its parent model

Syntax

```
void ssSetModelReferenceSampleTimeInheritanceRule(SimStruct *S,
  int_T rule)
```

Arguments

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

`rule`
: Rule for allowing submodels containing this S-function to inherit their sample times from the parent model.

Description

Use this macro in any callback from `mdlInitializeSizes` to `mdlSetWorkWidths` to specify the rule that determines whether submodels containing an S-function can inherit their sample times from their parent model. Use this macro only if your S-function inherits its sample time. If the S-function output is dependent on its inherited sample time, use the `DISALLOW_SAMPLE_TIME_INHERITANCE` rule to specify that submodels containing the S-function cannot inherit their sample times from their parent model. Otherwise, use the `USE_DEFAULT_FOR_DISCRETE_INHERITANCE` rule to allow sample time inheritance.

Languages

C, C++

Example

See “Creating Model Components” in the *Real-Time Workshop User’s Guide* for more information and examples that use this function.

See Also

`ssSetModelReferenceSampleTimeDefaultInheritance`,
`ssSetModelReferenceSampleTimeDisallowInheritance`

ssSetModeVectorValue

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++

Example

The following statement

```
ssSetModeVectorValue(S, 0, 1.0);
```

sets the first element of the mode vector to `1.0`.

See Also

`ssGetModeVectorValue`, `ssGetModeVector`

ssSetNumContStates

Purpose

Specify the number of continuous states that a block has

Syntax

```
void ssSetNumContStates(SimStruct *S, int_T n)
```

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 **Data Import/Export** pane of the Configuration Parameters dialog box.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/simomex.c` used in `sfcndemo_simomex.mdl`.

See Also

`ssSetNumDiscStates`, `ssGetNumContStates`

Purpose

Specify the number of discrete states that a block has

Syntax

```
void 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 **Data Import/Export** page of the Configuration Parameters dialog box.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/dsfunc.c` used in `sfcndemo_dsfunc.mdl`.

See Also

`ssSetNumContStates`, `ssGetNumDiscStates`

ssSetNumDWork

Purpose

Specify the number of data type work vectors used by a block

Syntax

```
boolean_T ssSetNumDWork(SimStruct *S, int_T nDWork)
```

Arguments

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

`nDWork`
: Number of data type work vectors.

Returns

The Boolean value `true` if `nDWork` is zero or a positive integer; otherwise, `false`.

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`. Use `ssSetDWorkName` to specify a name for the work vector.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl`.

See Also

`ssGetNumDWork`, `ssSetDWorkWidth`, `ssSetDWorkDataType`, `ssSetDWorkComplexSignal`, `ssSetDWorkName`

Purpose

Specify the number of input ports that a block has

Syntax

```
boolean_T ssSetNumInputPorts(SimStruct *S, int_T nInputPorts)
```

Arguments

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

`nInputPorts`
: Number of input ports on the block represented by `S`. Must be a nonnegative integer.

Returns

The Boolean value `true` if successful. Otherwise, returns `false`.

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 `false` if *`nInputPorts`* is negative or an error occurs while creating the ports.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

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. The elements of the IWork vector are initialized to `NULL` until values are assigned using `ssSetIWorkValue` or via the pointer obtained from`ssGetIWork`.

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also `ssSetNumRWork`, `ssSetNumPWork`, `ssSetIWorkValue`, `ssGetIWork`

Purpose

Specify the size of the block's mode vector

Syntax

`int_T ssSetNumModes(SimStruct *S, int_T 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`.

Returns

The number of modes specified by `nModes`, or `-1` if `DYNAMICALLY_SIZED`.

Description

Sets the size of the block's mode vector to `nModes`. The elements of the mode vector are initialized to `NULL` until values are assigned using `ssSetModeVectorValue` or via the pointer obtained from`ssGetModeVector`.

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. The Simulink engine 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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc.c` used in `sfcndemo_sfun_zc.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc_sat.c` used in `sfcndemo_sfun_zc_sat.mdl`.

See Also

`ssGetNumModes`, `ssGetModeVector`, `ssSetModeVectorValue`

Purpose

Specify the number of states for which a block detects zero crossings that occur between sample points

Syntax

```
int_T ssSetNumNonsampledZCs(SimStruct *S, int_T nNonsampledZCs)
```

Arguments

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

`nNonsampledZCs`
: Number of nonsampled zero crossings that a block detects.

Returns

The number of modes specified by `nNonsampledZCs`, or `-1` if `DYNAMICALLY_SIZED`.

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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc.c` used in `sfcndemo_sfun_zc.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_zc_sat.c` used in `sfcndemo_sfun_zc_sat.mdl`.

See Also

`ssSetNumModes`

Purpose

Specify the number of output ports that a block has

Syntax

```
boolean_T ssSetNumOutputPorts(SimStruct *S, int_T nOutputPorts)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nOutputPorts`
Number of output ports on the block represented by `S`. Must be a nonnegative integer.

Returns

The Boolean value `true` if successful. Otherwise, returns `false`.

Description

Use in `mdlInitializeSizes` to set the number of output ports to a nonnegative integer. Invoke the function 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, the Simulink engine displays an error message.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_port_constant.c` used in `sfcndemo_port_constant.mdl`.

See Also

`ssSetInputPortWidth`, `ssSetNumInputPorts`

Purpose

Specify the size of a block's pointer work vector

Syntax

```
int_T 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`.

Returns

The number of elements specified by `nPWork`, or `-1` if `DYNAMICALLY_SIZED`.

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. The elements of the pointer vector are initialized to `NULL` until values are assigned using `ssSetPWorkValue` or via the pointer obtained from`ssGetPWork`.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_rtwdwork.c` used in `sfcndemo_sfun_rtwdwork.mdl`.

See Also

`ssGetNumPWork`, `ssSetPWorkValue`, `ssGetPWork`

Purpose

Specify the number of run-time parameters created by this S-function

Syntax

```
void ssSetNumRunTimeParams(SimStruct *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. See “Run-Time Parameters” on page 8-8 for more information.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime2.c` used in `sfcndemo_runtime.mdl`.

See Also

`mdlSetWorkWidths`, `ssGetNumRunTimeParams`, `ssSetRunTimeParamInfo`

Purpose

Specify the size of a block's floating-point work vector

Syntax

`int_T ssSetNumRWork(SimStruct *S, int_T nRWork)`

Arguments

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

`nRWork`
: Number of elements in the floating-point work vector.

Returns

The number of elements specified by `nRWork`, or `-1` if `DYNAMICALLY_SIZED`.

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. The elements of the RWork vector are initialized to zero.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dynsize.c` used in `sfcndemo_sfun_dynsize.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfunmem.c` used in `sfcndemo_sfunmem.mdl`.

See Also

`ssSetNumIWork`, `ssSetNumPWork`

ssSetNumSampleTimes

Purpose

Specify the number of sample times that an S-Function block has

Syntax

`int_T ssSetNumSampleTimes(SimStruct *S, int_T nSampleTimes)`

Arguments

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

`nSampleTimes`
: Number of sample times that `S` has.

Returns

The number of sample times specified by `nSampleTimes`, or `-1` if `PORT_BASED_SAMPLE_TIMES`.

Description

Use in `mdlInitializeSizes` to set the number of sample times `S` has. Use a positive integer greater than 0 to set block-based sample times. Use `PORT_BASED_SAMPLE_TIMES` to set port-based sample times. See “Sample Times” on page 8-33 for more information.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/mixedmex.c` used in `sfcndemo_mixedmex.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_port_constant.c` used in `sfcndemo_port_constant.mdl`.

See Also

`ssGetNumSampleTimes`

Purpose

Specify the number of parameters that an S-Function block has

Syntax

```
int_T ssSetNumSFcnParams(SimStruct *S, int_T nSFcnParams)
```

Arguments

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

`nSFcnParams`
: Number of parameters that `S` has.

Returns

The number of parameters specified in `nSFcnParams`.

Description

Use in `mdlInitializeSizes` to set the number of expected S-function parameters.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvdtf.c` used in `sfcndemo_stvdtf.mdl`.

See Also

`ssGetNumSFcnParams`

ssSetOffsetTime

Purpose

Set the offset time of a block

Syntax

```
time_T ssSetOffsetTime(SimStruct *S, int_T st_index, time_T offset)
```

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`.

Returns

The `time_T` offset value specified by `offset`.

Description

Use this macro in `mdlInitializeSizes` or `mdlInitializeSampleTimes` to specify the offset of the sample time where *`st_index`* starts at 0. You must first invoke the macro `ssSetSampleTime` to set the sample time before assigning an offset. Otherwise, a continuous sample time is assumed and the offset is ignored.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/mixedmex.c` used in `sfcndemo_mixedmex.mdl`.

See Also

`ssGetOffsetTime`, `ssSetSampleTime`, `ssSetInputPortOffsetTime`, `ssSetOutputPortOffsetTime`

Purpose

Specify that an input port expects one-based indices

Syntax

```
void ssSetOneBasedIndexInputPort(SimStruct *S, int_T pIdx)
```

Arguments

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

`pIdx`
: Input port of the S-function.

Description

Use this macro in `mdlInitializeSizes` to specify that port `pIdx` expects one-based index values. By setting this macro, the Simulink software runs a diagnostic when it updates the diagram to check if the S-function input port expecting one-based indices is connected to a block that is producing zero-based indices. The Simulink software signals an error if it detects that the signal connected to this block is zero-based. Simulink blocks that can produce indices include the For Iterator and S-function blocks. If neither this macro nor `ssSetZeroBasedIndexInputPort` is invoked, the Simulink software does not run this diagnostic, even if the input port is connected to a block that produces indices.

Languages

C, C++

See Also

`mdlInitializeSizes`, `ssSetOneBasedIndexOutputPort`, `ssSetZeroBasedIndexInputPort`

ssSetOneBasedIndexOutputPort

Purpose

Specify that an output port emits one-based indices.

Syntax

```
void ssSetOneBasedIndexOutputPort(SimStruct *S, int_T pIdx)
```

Arguments

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

`pIdx`
: Output port of the S-function.

Description

Use this macro in `mdlInitializeSizes` to specify that port `pIdx` emits one-based index values. By setting this macro, the Simulink software runs a diagnostic when it updates the diagram to check if the S-function output port emitting one-based indices is connected to a block that expects zero-based indices. The Simulink software signals an error if it detects that the output port is connected to an input that expects zero-based indices. Simulink blocks that accept indices include the Selector, Assignment, and S-function blocks. If neither this macro nor `ssSetZeroBasedIndexOutputPort` is invoked, the Simulink software does not run this diagnostic, even if the output port is connected to a block that accepts indices.

Languages

C, C++

See Also

`mdlInitializeSizes`, `ssSetOneBasedIndexOutputPort`, `ssSetZeroBasedIndexOutputPort`

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. The options must be joined using the `OR` operator. For example:

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

See Chapter 12, "S-Function Options — Alphabetical List" for a description of the available options.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl`.

See Also

`mdlInitializeSizes`

ssSetOutputPortComplexSignal

Purpose

Set the numeric type (real or complex) of an output port

Syntax

```
CSignal_T ssSetOutputPortComplexSignal(SimStruct *S, input_T port,
 CSignal_T csig)
```

Arguments

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

`port`
: Index of an output port.

`csig`
: Numeric type of the signals emitted by `port`. Valid values are `COMPLEX_NO` (real signal), `COMPLEX_YES` (complex signal), and `COMPLEX_INHERITED` (dynamically determined).

Returns

`1` (`COMPLEX_YES`), `0` (`COMPLEX_NO`), or `-1` (`COMPLEX_INHERITED`) depending on the value specified by `csig`.

Description

Use this function in `mdlInitializeSizes` to initialize an output port numeric type. If the numeric type of the output port is determined dynamically, e.g., by a parameter setting, set the numeric type to `COMPLEX_INHERITED`. In this case, the S-function must provide `mdlSetOutputPortComplexSignal` and `mdlSetDefaultPortComplexSignals` methods to enable the numeric type to be set correctly during signal propagation. The default numeric type of an output port is `real`.

Languages

C, C++

Example

Assume that an S-function has three output ports. The first output port emits real (noncomplex) signals. The second output 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 the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sfun_cplx.c` used in `sfcndemo_cplx.mdl` and the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl` for complete examples that use this function.

See Also

`ssGetOutputPortComplexSignal`

ssSetOutputPortDataType

Purpose

Set the data type of an output port

Syntax

```
DTypeId ssSetOutputPortDataType(SimStruct *S, int_T port, DTypeId id)
```

Arguments

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

`port`
: Index of an output port.

`id`
: ID of the data type accepted by `port`.

Returns

The data type ID specified by `id`. Returns `-1` if `id` is `DYNAMICALLY_TYPED`.

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`. In this case, the S-function must provide `mdlSetOutputPortDataType` and `mdlSetDefaultPortDataTypes` methods to enable the data type to be set correctly during signal propagation.

For a list of the built-in data types, see `BuiltInDTypeID` in *`matlabroot`*`/simulink/include/simstruc_types.h`.

Note The data type of an output port is `double` (`real_T`) by default.

Languages

C, 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 the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl` and the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl` for complete examples that use this function.

See Also

`ssGetOutputPortDataType`

ssSetOutputPortDimensionInfo

Purpose

Specify information about the dimensionality of an output port

Syntax

```
int_T ssSetOutputPortDimensionInfo(SimStruct *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.

Returns

`1` if successful; otherwise, `0`.

Description

Specifies the dimension information for `port`. Use this function in `mdlInitializeSizes` to initialize the output port dimension information. 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`. In this case, the S-function must provide `mdlSetOutputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` methods to enable the signal dimensions to be set correctly during signal propagation.

Languages

C, C++

Example

The following example specifies that output port 0 emits 2-by-2 matrix signals.

```
  DECL_AND_INIT_DIMSINFO(di);
  int_T dims[2];
```

```
di.numDims = 2;
dims[0] = 2;
dims[1] = 2;
di.dims = dims;
di.width = 4;
ssSetOutputPortDimensionInfo(S,  0, &di);
```

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_matadd.c` used in `sfcndemo_matadd.mdl` for a complete example that uses this function.

See Also

`ssSetInputPortDimensionInfo`

ssSetOutputPortDimensionsMode

Purpose Sets the dimensions mode of the output port indexed by pIdx

Syntax

```
Void ssSetOutputPortDimensionsMode(SimStruct *S,int_T pIdx, DimensionsMoc
```

Arguments

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

`pIdx`
: Output port index being polled.

`mode`
: Enum value corresponding to the ports dimensions mode.

Returns No return value

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_concat1`

Purpose

Specify whether a port outputs framed data

Syntax

```
void ssSetOutputPortFrameData(SimStruct *S,  int_T port,
  Frame_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 `FRAME_INHERITED` (either frame or unframed output), `FRAME_NO` (unframed output only), and `FRAME_YES` (framed output only).

Description

Use in `mdlInitializeSizes` to specify whether an output port issues frame data only, unframed data only, or both. By default, output ports do not issue frame signals (`FRAME_NO`). If the S-function specifies `FRAME_INHERITED` for the frame status of any of its output ports, the S-function should include an `mdlSetInputPortFrameData` callback method to set the frame status for these ports.

The use of frame-based signals (`outputsFrames` has a value of `FRAME_YES`) requires a Signal Processing Blockset product license.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl`.

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`.

Returns `1` if successful; otherwise, `0`.

Description Use this function to specify that `port` emits an `m`-by-`n` matrix signal. If either dimension is `DYNAMICALLY_SIZED`, the other must be `DYNAMICALLY_SIZED` or `1`. If either dimension is dynamically sized, the S-function must provide `mdlSetOutputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` methods to enable the signal dimensions to be set correctly during signal propagation.

Languages C, C++

Example The following example specifies that output port 0 emits 2-by-2 matrix signals.

```
ssSetOutputPortMatrixDimensions(S,  0, 2, 2);
```

See the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sfun_frmad.c` used in `sfcndemo_frame.mdl` for a complete example that uses this function. Running this model requires a Signal Processing Blockset license.

See Also

`ssGetOutputPortDimensions`

ssSetOutputPortOffsetTime

Purpose

Specify the offset time of an output port

Syntax

```
real_T ssSetOutputPortOffsetTime(SimStruct *S, int_T outputPortIdx,
 real_T offset)
```

Arguments

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

`outputPortIdx`
: Index of the output port whose sample time is being set.

`offset`
: Offset time of an output port.

Returns

The `real_T` value of the offset time passed into the macro.

Description

Use in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify the sample time offset value for each output port index. Output port index numbers start at 0 and end at the total number of output ports minus 1. This should only be used if you have specified the S-function's sample times as port-based.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also

`ssSetNumOutputPorts`, `ssSetOutputPortSampleTime`

Purpose

Specify reusability of the memory allocated to the output port of an S-function

Syntax

```
void ssSetOutputPortOptimOpts(SimStruct *S, int_T port, uint_T val)
```

Arguments

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

`port`
: Index of an output port of `S`.

`val`
: Reusability of `port`. Permissible values are

- `SS_NOT_REUSABLE_AND_GLOBAL` (default value)
- `SS_REUSABLE_AND_LOCAL`
- `SS_REUSABLE_AND_GLOBAL`
- `SS_NOT_REUSABLE_AND_LOCAL`

Description

Use this macro to specify the reusability and scope of the memory allocated to an S-function output port. The reusability indicates whether or not the memory associated with the output port can be overwritten. You must specify that an output port is reusable if the output port connects to a Merge block. The scope indicates whether the model variables are stored locally or globally.

You cannot use `ssGetOutputPortSignal` or `ssGetOutputPortRealSignal` anywhere except in the `mdlOutputs` routine if you have specified that the output ports are reusable.

The Simulink product only uses the reusability setting during simulation. It disregards the memory scope setting, i.e., local or global, instead treating all S-function ports as global during simulation.

Note The Real-Time Workshop product uses the memory scope setting you specified when generating code from a model. If your S-function accesses the outputs only in `mdlOutputs`, the Real-Time Workshop product attempts to declare local variables for any outputs with a local scope. If your S-function uses the outputs in a way that precludes using a local scope, the generated code uses global variables for the outputs. See “Writing S-Functions for Multirate Multitasking Environments” in the *Real-Time Workshop User’s Guide* for more information.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also

`ssGetOutputPortOptimOpts`, `ssSetInputPortOptimOpts`, `ssSetInputPortOverWritable`

Purpose

Specify whether an output port can share its memory buffer with an input port

Syntax

```
void ssSetOutputPortOverwritesInputPort(SimStruct *S, int_T outIdx,
 int_T inIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`outIdx`
Index of the output port.

`inIdx`
Index of the input port.

Description

The argument `inIdx` tells the Simulink engine which input port of `S` can share its memory with the output port specified by `outIdx`. `inIdx` can have the following values:

- Index of an input port of `S` that can share its memory with the specified output port.

  You must use `ssSetInputPortOverWritable` to tell the engine that the specified input port can share its memory with an output port.

- `OVERWRITE_INPUT_ANY`

  The output port can share its memory with any input port.

- `OVERWRITE_INPUT_NONE`

  The output port must have its own memory buffer.

Languages

C, C++

See Also

`ssSetInputPortOverWritable`

ssSetOutputPortSampleTime

Purpose Specify the sample time of an output port

Syntax

```
real_T ssSetOutputPortSampleTime(SimStruct *S, int_T outputPortIdx,
 time_T 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 the output port.

Returns The `real_T` value of the sample time passed into the macro.

Description Use in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify the sample time period as continuous or as a discrete value for each output port index. Output port index numbers start at 0 and end at the total number of input ports minus 1. For a continuous sample time, specify `period` as `CONTINUOUS_SAMPLE_TIME`. To inherit the sample time, specify period as `INHERITED_SAMPLE_TIME`. You should use this macro only if you have specified port-based sample times.

If the S-function specifies `INHERITED_SAMPLE_TIME` for any of its ports, the S-function should include an `mdlSetOutputPortSampleTime` callback method. The callback method should set the sample time and offset of ports that inherit their sample time to the status that the Simulink engine assigns to them using its sample time propagation rules. The callback method can also assign the sample times and offsets of other ports on the block whose sample times are inherited.

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl`.

See Also `ssSetNumOutputPorts`, `ssSetOutputPortOffsetTime`

ssSetOutputPortVectorDimension

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 the vector or `DYNAMICALLY_SIZED`.

Returns

`1` if successful; otherwise, `0`.

Description

Specifies that `port` emits a `w`-element vector signal. If the width is dynamically sized, the S-function must provide `mdlSetOutputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` methods to enable the signal dimensions to be set correctly during signal propagation.

Note This function and the `ssSetOutputPortWidth` macro are functionally identical.

Languages

C, C++

Example

The following example specifies that output port 0 emits an 8-element matrix signal.

```
ssSetOutputPortVectorDimension(S, 0, 8);
```

See the S-function *matlabroot*`/toolbox/simulink/simdemos/simfeatures/src/sdotproduct.c` used in `sfcndemo_sdotproduct.mdl` for a complete example that uses this function.

See Also `ssSetOutputPortWidth`

ssSetOutputPortWidth

Purpose

Specify the width of an output port

Syntax

```
int_T ssSetOutputPortWidth(SimStruct *S, int_T port, int_T width)
```

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. If the width is dynamically sized, the S-function must provide `mdlSetOutputPortDimensionInfo` and `mdlSetDefaultPortDimensionInfo` methods to enable the signal dimensions to be set correctly during signal propagation.

Note This macro and `ssSetOutputPortVectorDimension` are functionally identical except that `ssSetOutputPortVectorDimension` returns 1 if successful, and 0 otherwise.

Languages

C, C++

Example

See the S-function `sfun_dtype_io.c` used in the `sfcndemo_dtype_io.mdl`.

See Also

`ssSetNumOutputPorts`, `ssSetInputPortWidth`, `ssSetOutputPortVectorDimension`

Purpose

Specify the name of the placement group of a block

Syntax

```
void ssSetPlacementGroup(SimStruct *S, const char_T *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 sorted order list for the model. There is no correlation between different placement groups. This macro should be invoked in `mdlInitializeSizes`.

Note This macro is typically used to create device driver blocks.

Languages

C, 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.

Returns

The pointer passed into the macro.

Description

Sets the `idx` element of the S-function'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.

Languages

C, C++

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.

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.

Returns

The `real_T` value passed into the macro.

Description

Sets the `idx` element of the S-function'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.

Languages

C, C++

Example

The following statement

```
ssSetRWorkValue(S, 0, 1.0);
```

sets the first element of the work vector to `1.0`.

See Also

`ssGetNumRWork`, `ssGetRWork`, `ssGetRWorkValue`

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.

`info`
: `ssParamRec` structure containing the attributes of the run-time parameter.

Description

Use this function in `mdlSetWorkWidths` to specify information about a run-time parameter. Use an `ssParamRec` structure to pass the parameter attributes to the function. See “Run-Time Parameters” on page 8-8 for more information.

ssParamRec Structure

The `simstruc.h` macro defines this structure as follows:

```
typedef struct ssParamRec_tag {
    const char *name;
    int_T      nDimensions;
    int_T      *dimensions;
    DTypeId    dataTypeId;
    boolean_T  complexSignal;
    void       *data;
    const void *dataAttributes;
    int_T      nDlgParamIndices;
    int_T      *dlgParamIndices;
    TransformedFlag transformed; /* Transformed status */
    boolean_T  outputAsMatrix;   /* Write out parameter
                                  * as a vector (false)
                                  * [default] or a matrix (true)
                                  */
} ssParamRec;
```

The record contains the following fields.

`name`
: Name of the parameter. This must point to persistent memory. Do not set to a local variable (`static char name[32]` or strings name are okay).

`nDimensions`
: Number of dimensions that this parameter has.

`dimensions`
: Array giving the size of each dimension of the parameter.

`dataTypeId`
: Data type of the parameter. For a list of 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.

Note `ssSetRunTimeParamInfo` must set this field to the parameter's actual value. This is necessary for Real–Time Workshop to perform the parameter pooling optimization correctly. If you fail to set the data field at the time of registration and then fill it in at a later juncture, you might see an error indicating that some parameters were incorrectly pooled with each other.

`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, the Real-Time Workshop product 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 product 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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime4.c` used in `sfcndemo_runtime.mdl`.

See Also

`mdlSetWorkWidths`, `mdlProcessParameters`, `ssGetNumRunTimeParams`, `ssGetRunTimeParamInfo`

Purpose

Set the period of a sample time

Syntax

```
time_T ssSetSampleTime(SimStruct *S, int_T st_index, time_T period)
```

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`.

Returns

The `time_T` sample time value specified by `period`.

Description

Use this macro in `mdlInitializeSizes` or `mdlInitializeSampleTimes` to specify the period of the sample time where *`st_index`* starts at 0. See "Setting Sample Times and Offsets" on page 1-16 for more information. Use `ssSetOffsetTime` if the sample time has an associated offset.

Languages

C, C++

Example

See the following S-functions for examples that use this function:

- *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/mixedmex.c` used in `sfcndemo_mixedmex.mdl`
- *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dtype_io.c` used in `sfcndemo_dtype_io.mdl`
- *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvdtf.c` used in `sfcndemo_stvdtf.mdl`

See Also

`ssSetInputPortSampleTime`, `ssSetOutputPortSampleTime`, `ssSetOffsetTime`

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 Simulink product versions. Use `ssSetSFcnParamTunable` to set tunability of S-function parameters in new S-functions.

Languages

C, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_multirate.c` used in `sfcndemo_sfun_multirate.mdl` and the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_port_trigger` used in `sfcndemo_port_triggered.mdl`.

See Also

`ssSetSFcnParamTunable`, `ssGetSFcnParamsCount`

ssSetSFcnParamTunable

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 `SS_PRM_TUNABLE` (`true` / tunable), `SS_PRM_NOT_TUNABLE` (`false` / not tunable), or `SS_PRM_SIM_ONLY_TUNABLE` (tunable only during simulation).

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.

If you specify the `SS_PRM_TUNABLE` option, you must create a corresponding run-time parameter (see "Creating Run-Time Parameters" on page 8-9). You do not have to create a corresponding run-time parameter if you specify the `SS_PRM_SIM_ONLY_TUNABLE` option.

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`, the Simulink engine issues the diagnostic whenever it encounters an S-function that fails to specify the tunability of all its parameters.

Languages

C, C++

Example

See the following S-functions for examples that use this function:

- *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_atol.c` used in `sfcndemo_sfun_atol.mdl`
- *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvctf.c` used in `sfcndemo_stvctf.mdl`
- *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/stvdtf.c` used in `sfcndemo_stvdtf.mdl`

See Also

`ssGetSFcnParamsCount`

ssSetSignalSizesComputeType

Purpose Set the type of output dependency on the input signal.

Syntax `void ssSetSignalSizesComputeType (SimStruct *S, SS_VariableSizeComputeTyp`

Arguments

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

`Type`
: Enum value corresponding to the signal size compute type..

Returns No return value

Description Use this function in mdlSetWorkWidths if any output port dimensions mode is VARIABLE_DIMS_MODE to set the signal size type used to obtain the output signal size. Possible types are SS_VARIABLE_SIZE_FROM_INPUT_SIZE , i.e. the output sizes only depend on input sizes SS_VARIABLE_SIZE_FROM_INPUT_VALUE_AND_SIZE, i.e., the output sizes depend on input values

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_varsize_concat1`

Purpose

Specify the behavior of a Simulink S-function when saving and restoring the SimState of a model containing the S-function.

Syntax

```
void ssSetSimStateCompliance(S, setting)
```

Arguments

`S`
: SimStruct representing an S-function block.

`setting`
: Define how to treat an S-function simulation state when saving and restoring the model simulation state. Permissible values are:

- `SIM_STATE_COMPLIANCE_UNKNOWN` (default value)
- `USE_DEFAULT_SIM_STATE`
- `HAS_NO_SIM_STATE`
- `DISALLOW_SIM_STATE`
- `USE_CUSTOM_SIM_STATE`

Description

This function allows you to specify how Simulink should treat an S-function during a save and restore of the model simulation state (SimState). If an S-functions does not specify its `SimStateCompliance`, then Simulink assumes the setting `SIM_STATE_COMPLIANCE_UNKNOWN`. This setting instructs Simulink to issue a warning and then switch to `USE_DEFAULT_SIM_STATE` in lieu of the `SIM_STATE_COMPLIANCE_UNKNOWN`. If the option is set to `USE_DEFAULT_SIM_STATE` and if the S-function does not use `PWorks`, then Simulink treats the S-function like a built-in block. Simulink saves and restores the same data as the SimState (e.g., continuous states and work vectors), as it would for a built-in block.

Languages

C, C++

Example

The following example uses this function to specify the simulation state compliance of an S-function in the `mdlInitializeSizes` method. The specification is based upon the first string parameter value.

```
static void mdlInitializeSizes(SimStruct* S)
{
    ssSetNumSFcnParams(S, 2); /* two parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
   return;
    ssSetSFcnParamTunable(S, 0, false);
    ssSetSFcnParamTunable(S, 1, false);

    {
        boolean_T visibility = 0U;
        ssSimStateCompliance setting =
            GetSimSnapParameterSetting(S, &visibility);
        if (ssGetErrorStatus(S)) return;

        ssSetSimStateCompliance(S, setting);
        ssSetSimStateVisibility(S, visibility);
    }
```

See the full source code (`/toolbox/simulink/simdemos/simfeatures/src sfun_simstate.c`) for an example of how to get the S-function's simulation compliance.

See Also

`ssSetSimStateVisibility`

Purpose

Specify whether to make the S-function's simulation state visible in the simulation state of the model.

Syntax

```
void ssSetSimStateVisibility(S, visibility)
```

Arguments

`S`
: SimStruct representing an S-function block.

`visibility`
: Option to specify the visibility of the S-function simulation state. The default is `false`; the simulation state is hidden.

Description

This function allows you to specify whether or not the simulation state of the S-function is accessible from the simulation state of the model. When this option is set to `true`, you can access the SimState of this block via the `getBlockSimState` method of `ModelSimState` and you can restore any modified values via the `setBlockSimState` of the `ModelSimState`.

Languages

C, C++

Example

The following example uses this function to specify whether the simulation state of an S-function should be visible in the simulation state of the model. The specification is based upon the second (boolean) parameter value.

```
static void mdlInitializeSizes(SimStruct* S)
{
    ssSetNumSFcnParams(S, 2); /* two parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
   return;
    ssSetSFcnParamTunable(S, 0, false);
    ssSetSFcnParamTunable(S, 1, false);

    {
        boolean_T visibility = 0U;
        ssSimStateCompliance setting =
            GetSimSnapParameterSetting(S, &visibility);
        if (ssGetErrorStatus(S)) return;

        ssSetSimStateCompliance(S, setting);
        ssSetSimStateVisibility(S, visibility);
    }
```

See `/toolbox/simulink/simdemos/simfeatures/src` `sfun_simstate.c` for the full source code.

See Also

`ssSetSimStateCompliance`

Purpose

Ask the Simulink engine 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, C++

Example

The following example uses this macro to ask the Simulink engine 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 the Simulink engine 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.

See Also

ssIsVariableStepSolver

Purpose

Set the simulation stop requested flag

Syntax

```
void ssSetStopRequested(SimStruct *S, int_T 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 1, the Simulink engine halts the simulation at the end of the current time step.

Languages

C, 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, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/vsfunc.c` used in `sfcndemo_vsfunc.mdl`.

See Also `ssGetTNext`, `ssGetT`, `mdlGetTimeOfNextVarHit`

Purpose

Specify user data

Syntax

```
void ssSetUserData(SimStruct *S, void *data)
```

Arguments

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

`data`
: User data.

Description

Stores a pointer to the memory location containing the S-function's user data. To avoid memory leaks, the S-function must free this memory location during the call to `mdlTerminate`.

An S-function containing user data must perform the following steps.

1. Allocate memory for the user data, using a customized structure to store more complicated data.
2. Set the `SS_OPTION_CALL_TERMINATE_ON_EXIT` option in `mdlInitializeSizes`, to ensure the Simulink engine always calls the `mdlTerminate` function.
3. Store the pointer to the memory location in the user data, using a call to `ssSetUserData`.
4. In `mdlTerminate`, use `ssGetUserData` to retrieve the pointer to the memory location and free the memory.

See "Creating Run-Time Parameters from Multiple S-Function Parameters" on page 8-12 for an example that uses user data in conjunction with run-time parameters.

Note You cannot access user data from within a Target Language Compiler (TLC) file. Use DWork vectors instead of user data if you need to write a TLC-file to inline the S-function during code generation. See `sfcndemo_sfun_rtwdwork.mdl` for an example using DWork vectors.

Languages C, C++

Example See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime4.c` used in `sfcndemo_runtime.mdl`.

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. The Simulink engine 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, C++

Example

The following statement

```
ssSetVectorMode(S, SS_1_D_OR_ROW_VECT);
```

indicates that the S-function supports row-vectors for the input and output port signals.

Purpose Specify that an input port expects zero-based indices

Syntax

```
void ssSetZeroBasedIndexInputPort(SimStruct *S, int_T pIdx)
```

Arguments

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

`pIdx`
: Input port of the S-function.

Description Use this macro in `mdlInitializeSizes` to specify that port `pIdx` expects zero-based index values. By setting this macro, the Simulink engine runs a diagnostic when it updates the diagram to check if the S-function input port expecting zero-based indices is connected to a block that is producing one-based indices. The engine signals an error if it detects that the signal connected to this block is one-based. Simulink blocks that can produce indices include the For Iterator and S-function blocks. If neither this macro nor `ssSetOneBasedIndexInputPort` is invoked, The engine does not run this diagnostic, even if the input port is connected to a block that produces indices.

Languages C, C++

See Also `mdlInitializeSizes`, `ssSetZeroBasedIndexOutputPort`, `ssSetOneBasedIndexInputPort`

ssSetZeroBasedIndexOutputPort

Purpose Specify that an output port emits zero-based indices.

Syntax

```
void ssSetZeroBasedIndexOutputPort(SimStruct *S, int_T pIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`pIdx`
Output port of the S-function.

Description Use this macro in `mdlInitializeSizes` to specify that port `pIdx` emits zero-based index values. By setting this macro, the Simulink engine runs a diagnostic when it updates the diagram to check if the S-function output port emitting zero-based indices is connected to a block that expects one-based indices. The engine signals an error if it detects that the output port is connected to an input that expects one-based indices. Simulink blocks that accept indices include the Selector, Assignment, and S-function blocks. If neither this macro nor `ssSetOneBasedIndexOutputPort` is invoked, the engine does not run this diagnostic, even if the output port is connected to a block that accepts indices.

Languages C, C++

See Also `mdlInitializeSizes`, `ssSetZeroBasedIndexInputPort`, `ssSetOneBasedIndexOutputPort`

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`

SimStruct representing an S-Function block.

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, C++

Example

See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime1.c` used in `sfcndemo_runtime.mdl`.

See Also

`mdlProcessParameters`, `ssUpdateRunTimeParamInfo`, `ssRegAllTunableParamsAsRunTimeParams`

ssUpdateRunTimeParamData

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 the 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, C++

Example

See the S-functions `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime2.c` and `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime4.c`, both used in `sfcndemo_runtime.mdl`.

See Also

`mdlProcessParameters`, `ssGetRunTimeParamInfo`, `ssUpdateAllTunableParamsAsRunTimeParams`, `ssRegAllTunableParamsAsRunTimeParams`

Purpose

Update a run-time parameter that corresponds to a dialog parameter

Syntax

```
void ssUpdateDlgParamAsRunTimeParam(SimStruct *S, int_T rtIdx)
```

Arguments

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

`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 its corresponding dialog parameter. Use `ssRegDlgParamAsRunTimeParam` to associate a dialog parameter with a run-time parameter. If necessary, `ssUpdateDlgParamAsRunTimeParam` converts the data type of the dialog parameter value to the data type specified by `dtId`, the data type ID specified in `ssRegDlgParamAsRunTimeParam`.

Languages

C, C++

Example

See the S-function `*matlabroot*/toolbox/simulink/simdemos/simfeatures/src/sfun_runtime3.c` used in `sfcndemo_runtime.mdl`.

See Also

`ssUpdateAllTunableParamsAsRunTimeParams`, `ssRegDlgParamAsRunTimeParam`

ssUpdateRunTimeParamInfo

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 modify the parameter's attributes record and update the run-time parameter, using this macro.

Note If you used `ssRegAllTunableParamsAsRunTimeParams` to create the run-time parameters, use `ssUpdateAllTunableParamsAsRunTimeParams` to update the parameters.

Languages

C, C++

See Also

`mdlProcessParameters`, `ssGetRunTimeParamInfo`, `ssUpdateAllTunableParamsAsRunTimeParams`, `ssRegAllTunableParamsAsRunTimeParams`

Purpose Display a warning message

Syntax `void ssWarning(SimStruct *S, const char_T *msg)`

Arguments

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

`msg`
: Warning message.

Description Displays `msg`. This macro expands to `mexWarnMsgTxt` when compiled for use with the Simulink product. When compiled for use with the Real-Time Workshop product, the macro 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, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfun_dynsize.c` used in `sfcndemo_sfun_dynsize.mdl`.

See Also `ssSetErrorStatus`, `ssPrintf`

ssWriteRTW2dMatParam

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 11-312).

`nRows`
: Number of rows in the matrix.

`nCols`
: Number of columns in the matrix.

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description

Use this function in `mdlRTW` to write a numeric matrix parameter to this S-function’s *model*.rtw file.

Languages

C, C++

See Also

`mdlRTW`

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 *cValue, 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.

`cValue`
: Imaginary elements of the parameter array.

`dataType`
: Data type of the parameter elements (see “Specifying Data Type Info” on page 11-312).

`nRows`
: Number of rows in the matrix.

`nCols`
: Number of columns in the matrix.

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description

Use this function in `mdlRTW` to write a matrix parameter in MATLAB format to this S-function’s *`model`*`.rtw` file.

Languages

C, 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 *cValue, 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 11-312).

size
: Number of elements in the vector.

Returns An int_T (1 or 0) or boolean_T (true or false) indicating the success or failure of the function.

Description Use this function in mdlRTW to write a vector parameter in MATLAB format to this S-function's *model*.rtw file.

Languages C, C++

See Also mdlRTW, ssWriteRTWMxVectParam

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" on page 11-310).

`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" on page 11-310).

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

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.

Note This function is provided for compatibility with S-functions that do not use run-time parameters and will be deprecated in future releases. It is suggested that you use run-time parameters (see "Run-Time Parameters" on page 8-8). 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.

- `SSWRITE_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

Argument	Description
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 11-312)

- 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
int_T vectLen	Length of the vector
int_T dtInfo	Data type of the vector (see “Specifying Data Type Info” on page 11-312)

- 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.

Languages

C, C++

Example

See the S-function /toolbox/simulink/simdemos/simfeatures/src/sfun_multiport.c used in sfcndemo_sfun_multiport.mdl.

See Also

mdlRTW

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 11-313).

`settingName`
: Name of the parameter.

`...`
: Remaining arguments depend on the parameter type (see "Parameter Setting Type-Specific Arguments" on page 11-313).

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description

Use this function in `mdlRTW` to write nontunable parameter setting information to this S-function's *model*.rtw file. You can also use this macro to write out other constant values required to generate code for this S-function.

Parameter Setting Type-Specific Arguments

This section lists the parameter-specific arguments required by each parameter type.

- `SSWRITE_VALUE_STR` (unquoted string): Use `SSWRITE_VALUE_STR` for any single word string that does not begin with a number or contain mixed characters. In some cases, `ssWriteRTWParamSettings` encloses the string in quotation marks even though you used

SSWRITE_VALUE_STR. For example, if the length of the string causes the corresponding line in the MDL-file to wrap in the middle of the string, ssWriteRTWParamSettings uses quotation marks to maintain the value of the string.

Argument	Description
`const char_T *value`	String (e.g., USA)

- SSWRITE_VALUE_QSTR (quoted string): Use SSWRITE_VALUE_QSTR for any multi-word or mixed character string or for single word strings that begin with a number. Code generation with the Real-Time Workshop product errors out if any string beginning with a number is not enclosed in quotation mark.

Argument	Description
`const char_T *value`	String (e.g., "U.S.A.")

- SSWRITE_VALUE_VECT_STR (vector of strings): Use SSWRITE_VALUE_VECT_STR to write a vector of strings with each element enclosed in quotation marks.

Argument	Description
`const char_T *value`	Vector of strings (e.g., ["USA", "Mexico"])
`int_T nItemsInVect`	Size of the vector

- SSWRITE_VALUE_NUM (number): Use SSWRITE_VALUE_NUM to a write real, floating-point value.

Argument	Description
`const real_T value`	Number (e.g., 2)

- SSWRITE_VALUE_VECT (vector of numbers): Use SSWRITE_VALUE_VECT to a write real, floating-point vector of values.

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): Use `SSWRITE_VALUE_2DMAT` to a write real, floating-point 2-D matrix of values.

Argument	Description
`const real_T *value`	Matrix of numbers (e.g., `[[170, 130],[60, 40]]`)
`int_T nRows`	Number of rows in the matrix
`int_T nCols`	Number of columns in the matrix

- `SSWRITE_VALUE_DTYPE_NUM` (data-typed number): Use `SSWRITE_VALUE_DTYPE_NUM` to write a complex value in Simulink format, or with a data type other than double.

Argument	Description
`const void *value`	Number (e.g., `[3+4i]`)
`int_T dtInfo`	Data type (see "Specifying Data Type Info" on page 11-312)

- `SSWRITE_VALUE_DTYPE_VECT` (data-typed vector): Use `SSWRITE_VALUE_DTYPE_VECT` to write a complex vector of values in Simulink format, or with a data type other than double.

Argument	Description
`const void *value`	Data-typed vector (e.g., `[1+2i, 3+4i]`)

Argument	Description
`int_T vectLen`	Size of the vector
`int_T dtInfo`	Data type (see "Specifying Data Type Info" on page 11-312)

- `SSWRITE_VALUE_DTYPE_2DMAT` (data-typed matrix): Use `SSWRITE_VALUE_DTYPE_2DMAT` to write a complex 2-D matrix of values in Simulink format, or with a data type other than double.

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 11-312)

- `SSWRITE_VALUE_DTYPE_ML_VECTOR` (data-typed MATLAB vector): Use `SSWRITE_VALUE_DTYPE_ML_VECTOR` to write the real and imaginary parts of a complex vector of values as separate arrays. `SSWRITE_VALUE_DTYPE_ML_VECTOR` allows you to specify a data type for the 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]`)

Argument	Description
`int_T vectLen`	Number of elements in the vector
`int_T dtInfo`	Data type (see "Specifying Data Type Info" on page 11-312)

- `SSWRITE_VALUE_DTYPE_ML_2DMAT` (data-typed MATLAB matrix): Use `SSWRITE_VALUE_DTYPE_ML_2DMAT` to write the real and imaginary parts of a complex 2-D matrix of values as separate matrices. `SSWRITE_VALUE_DTYPE_ML_2DMAT` allows you to specify a data type for the values.

Argument	Description
`const void *RValue`	Real component of the matrix (e.g., `[1 5 3 6]`)
`const void *IValue`	Imaginary 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 11-312)

Languages

C, C++

Example

See the S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/src/sfun_frmad.c` used in `sfcndemo_frame.mdl` for a complete example that uses this function. Running this model requires a Signal Processing Blockset license.

This S-function begins with the following variable declarations associated with the S-function parameters.

```
/* Parameters */
enum {FCN_ARGC = 0, AMP_ARGC,     FREQ_ARGC,     TS_ARGC,
      FRMSIZE_ARGC, NOISAMP_ARGC, NOISFREQ_ARGC, NUM_ARGS};

#define FCN_ARG(S)      (ssGetSFcnParam(S,FCN_ARGC))
#define AMP_ARG(S)      (ssGetSFcnParam(S,AMP_ARGC))
#define FREQ_ARG(S)     (ssGetSFcnParam(S,FREQ_ARGC))
#define TS_ARG(S)       (ssGetSFcnParam(S,TS_ARGC))
#define FRMSIZE_ARG(S)  (ssGetSFcnParam(S,FRMSIZE_ARGC))
#define NOISAMP_ARG(S)  (ssGetSFcnParam(S,NOISAMP_ARGC))
#define NOISFREQ_ARG(S) (ssGetSFcnParam(S,NOISFREQ_ARGC))

#define GET_FRMSIZE(S)  (mxGetPr(FRMSIZE_ARG(S)))[0]
```

The S-function's `mdlRTW` function then uses `ssWriteRTWParamSettings` to write the S-function parameters to the *`model`*`.rtw` file.

```
real_T  noisA = mxGetPr(NOISAMP_ARG(S))[0];
real_T  noisF = mxGetPr(NOISFREQ_ARG(S))[0];
real_T  ts    = mxGetPr(TS_ARG(S))[0];
int_T   fcn   = (int) (mxGetPr(FCN_ARG(S))[0]);
int32_T fsize = mxGetPr(FRMSIZE_ARG(S))[0];

if (!ssWriteRTWParamSettings(S, 5,
     SSWRITE_VALUE_STR, "Function", (fcn == 1) ? "Constant" : "Sine",
     SSWRITE_VALUE_NUM, "Ts",       ts,
     SSWRITE_VALUE_DTYPE_NUM, "FrameSize", &fsize,
         DTINFO(SS_INT32, COMPLEX_NO),
     SSWRITE_VALUE_NUM, "NoiseAmp",  noisA,
     SSWRITE_VALUE_NUM, "NoiseFreq", noisF)) {
        return; /* An error occurred. */
    }
```

When code is generated for the model, the Real-Time Workshop product first writes a structure named `SFcnParamSettings` to the *`model`*`.rtw` file based on the information provided in the S-function's `mdlRTW` method. In this example, the resulting `SFcnParamSettings` is:

```
SFcnParamSettings {
 Function  Sine
 Ts        0.005
 FrameSize 64
 NoiseAmp  4.0
 NoiseFreq 80.0
      }
```

The S-function's Target Language Compiler file *`matlabroot`*`/toolbox/simulink/blocks/tlc_c/sfun_frmad.tlc` then accesses the S-function parameters using the variable names in the `SFcnParamSettings` structure. For example:

```
%assign fnName   = SFcnParamSettings.Function
%assign frmSize  = SFcnParamSettings.FrameSize
%assign ts       = SFcnParamSettings.Ts
%assign noisA    = SFcnParamSettings.NoiseAmp
%assign noisF    = SFcnParamSettings.NoiseFreq
```

See Also

`mdlRTW`

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 the Simulink built-in data types (see BuiltInDTypeId in simstruc_types.h in the *matlabroot*/simulink/include subdirectory) or the ID of a user-defined type (see "Custom Data Types" on page 8-29).

Returns

An int_T (1 or 0) or boolean_T (true or false) indicating the success or failure of the function.

Description

Use this function in mdlRTW to write scalar parameters to this S-function's *model*.rtw file.

Languages

C, 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.

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description

Use this function in `mdlRTW` to write strings to this S-function's *`model`*`.rtw` file.

Languages

C, C++

See Also

`mdlRTW`

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.

Returns An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description Use this function in `mdlRTW` to write string parameters to this S-function's *`model`*`.rtw` file. This function returns `1` (`true`) if successful. The output of this macro can also be correctly assigned to `boolean_T`.

Languages C, C++

See Also `mdlRTW`

Purpose

Write a vector of string parameters 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.

Returns

An `int_T` (1 or 0) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description

Use this function in `mdlRTW` to write a vector of string parameters to this S-function's *model*.rtw file.

Languages

C, C++

Example

The following lines write the three strings `one` , `two` , and `three` to a parameter named `count` in the *model*.rtw file. To create the vector of strings to pass to the `ssWriteRTWStrVectParam` function, enclosed the entire string in quotation marks and square brackets. Then, enclose each individual string parameter in quotation marks preceded by a backslash, as follows:

```
const char *str = "[\"one\",\"two\",\"three\"]";

if (!ssWriteRTWStrVectParam(S, "count", str,3)){
    return;
}
```

When code is generated for a model containing this S-function, the *model*.rtw file contains the line:

```
count        ["one","two","three"]
```

You can access the elements of the parameter `count` in the S-function's TLC file. For example, the following line in the TLC file:

```
/* Loop number = %<count[0]> */
```

appears as the following comment in the generated code:

```
/* Loop number = one */
```

See Also

`mdlRTW`

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 11-312).

`size`
: Number of elements in the vector.

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description

Use this function in `mdlRTW` to write a vector parameter in Simulink format to this S-function’s *`model`*`.rtw` file.

Languages

C, C++

See Also

`mdlRTW`, `ssWriteRTWMxVectParam`

ssWriteRTWWorkVect

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 the work vector (must be `RWork`, `IWork`, or `PWork`).

`nNames`
: Number of names (see the next argument).

`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).

Returns

An `int_T` (`1` or `0`) or `boolean_T` (`true` or `false`) indicating the success or failure of the function.

Description

Use this function in `mdlRTW` to write work vectors to this S-function's *`model`*`.rtw` file. For example:

```
static void mdlRTW(SimStruct *S)
{
    if (!ssWriteRTWWorkVect(S, "RWork", 1 /* nNames */,
                            "InputAtLastUpdate", ssGetNumRWork(S))) {
        return;
    }
    /*
      This registration of the symbol "InputAtLastUpdate"
   allows sfunmem.tlc to call
   LibBlockRWork(InputAtLastUpdate,[...])
```

```
         */

    }
```

Languages C, C++

Example See the S-function *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/sfunmem.c` used in `sfcndemo_sfunmem.mdl`.

See Also `mdlRTW`

S-Function Options — Alphabetical List

This section describes the S-function options available through `ssSetOptions`. Each S-function sets its applicable options at the end of its `mdlInitializeSizes` method. Use the `OR` operator (`|`) to set multiple options. For example:

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

SS_OPTION_ALLOW_CONSTANT_PORT_SAMPLE_TIME
SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION
SS_OPTION_ALLOW_PARTIAL_DIMENSIONS_CALL
SS_OPTION_ALLOW_PORT_SAMPLE_TIME_IN_TRIGSS
SS_OPTION_ASYNC_RATE_TRANSITION
SS_OPTION_ASYNCHRONOUS
SS_OPTION_CALL_TERMINATE_ON_EXIT
SS_OPTION_CAN_BE_CALLED_CONDITIONALLY
SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME
SS_OPTION_DISCRETE_VALUED_OUTPUT
SS_OPTION_EXCEPTION_FREE_CODE
SS_OPTION_FORCE_NONINLINED_FCNCALL
SS_OPTION_NONVOLATILE
SS_OPTION_PLACE_ASAP
SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED
SS_OPTION_REQ_INPUT_SAMPLE_TIME_MATCH
SS_OPTION_RUNTIME_EXCEPTION_FREE_CODE
SS_OPTION_SIM_VIEWING_DEVICE
SS_OPTION_SFUNCTION_INLINED_FOR_RTW

SS_OPTION_SUPPORTS_ALIAS_DATA_TYPES
SS_OPTION_USE_TLC_WITH_ACCELERATOR
SS_OPTION_WORKS_WITH_CODE_REUSE

SS_OPTION_ALLOW_CONSTANT_PORT_SAMPLE_TIME

Purpose Allow constant sample time for a port

Description Allows an S-function with port-based sample times to specify or inherit constant sample times. Setting this option tells the Simulink engine that all input and output ports support constant sample times. See "Specifying Constant Sample Time for a Port" on page 8-40 for more information.

Example See `sfun_port_constant.c`, the source file for the `sfcndemo_port_constant.mdl` demo, for an example.

See Also `SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME`

SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION

Purpose

Allow scalar expansion of input ports

Description

Specifies that the input to your S-function input ports can be have a width of either 1 or the size specified by the port, usually referred to as the block width. The S-function expands scalar inputs to the same dimensions as the block width. See “Scalar Expansion of Inputs” on page 8-26 for more information.

Example

See `sfun_multiport.c`, the source file for the `sfcndemo_sfun_multiport.mdl` demo, for an example.

Purpose

Allow calls to `mdlSetInputPortDimensionInfo` and `mdlSetOutputPortDimensionInfo` with partial dimension information

Description

Indicates the S-function can handle dynamically dimensioned signals. By default, the Simulink engine calls the `mdlSetInputPortDimensionInfo` or `mdlSetOutputPortDimensionInfo` methods if the number of dimensions and size of each dimension for the candidate port are fully known. If `SS_OPTION_ALLOW_PARTIAL_DIMENSIONS_CALLS` is set, the engine may also call these methods with partial dimension information. For example, the methods may be called when the port width is known, but the actual 2-D dimensions are unknown. See `mdlSetDefaultPortDimensionInfo` for more information.

See Also

`mdlSetDefaultPortDimensionInfo`

SS_OPTION_ALLOW_PORT_SAMPLE_TIME_IN_TRIGSS

Purpose Allow an S-function with port-based sample times to operate in a triggered subsystem

Description Allows an S-function that uses port-based sample times to operate in a triggered subsystem. During sample time propagation, use the macro `ssSampleAndOffsetAreTriggered` to determine if the sample and offset times correspond to the block being in a triggered subsystem. If the block is triggered, all port sample times must be either triggered or constant. See "Configuring Port-Based Sample Times for Use in Triggered Subsystems" on page 8-42 for more information.

Example See `sfun_port_triggered.c`, the source file for the `sfcndemo_port_triggered.mdl` demo, for an example.

See Also `ssSampleAndOffsetAreTriggered`

Purpose

Create a read-write pair of blocks that ensure correct data transfer

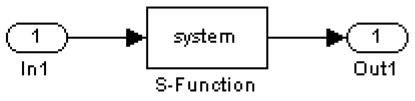
Description

Creates a read-write pair of blocks intended to guarantee correct data transfers between a synchronously (periodic) and an asynchronously executing subsystem or between two asynchronously executing subsystems. Both the read S-function and write S-function should set this option.

An asynchronously executed function-call subsystem is a function-call subsystem driven by an S-function with the `SS_OPTION_ASYNCHRONOUS` specified.

The Simulink engine defines two classes of asynchronous rate transitions.

- Read-write pairs. In this class, two blocks, using a technique such as double buffering, ensure data integrity in a multitasking environment. When creating the read-write pair of blocks, the S-functions for these blocks should set the `SS_OPTION_ASYNC_RATE_TRANSITION` option. Furthermore, the `MaskType` property of the read block, must include the string `read` and the `MaskType` property of write block must include the string `write`.
- A single protected or unprotected block. To create a single Protected Rate Transition block, create a subsystem that contains the following

  and set the `Tag` value of the Outport block to `AsyncRateTransition`. The S-function then provides the code for the protected transition. Note, this S-function does not set the `SS_OPTION_ASYNC_RATE_TRANSITION` option.

See Also

`SS_OPTION_ASYNCHRONOUS`

SS_OPTION_ASYNCHRONOUS

Purpose

Specify this S-function drives a function-call subsystem attached to interrupt service routines

Description

Specifies that the S-function is driving function-call subsystems attached to interrupt service routines. This option applies only to S-functions that have no input ports during code generation and 1 output port. During simulation, the S-function may have an input port to provide a condition on which to execute. 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. Specifying this option

- Informs the Simulink engine that there is no implied data dependency involving the data sources or destinations of the function-call subsystem called by the S-function.
- Causes the function-call subsystem attached to the S-function to be colored cyan, indicating that it does not execute at a periodic rate.
- Enables additional checks to verify that the model is constructed correctly.

  1. The engine validates that the appropriate asynchronous rate transition blocks reside between the cyan function-call subsystem. The engine also checks that period tasks exists. You can directly read and write from the function-call subsystem by using a block that has no computational overhead. To ensure safe task transitions between period and asynchronous tasks, use the `SS_OPTION_ASYNC_RATE_TRANSITION` option.
  2. For data transfers between two asynchronously executed (cyan) function-call subsystem, the engine validates that the appropriate asynchronous task transition blocks exits.

See Also

`SS_OPTION_ASYNC_RATE_TRANSITION`

Purpose Force call to `mdlTerminate`

Description Guarantees the Simulink engine calls the S-function's `mdlTerminate` method before destroying a block that references the S-function. Calling `mdlTerminate` allows your S-function to clean up after itself, for example, by freeing memory it allocated during a simulation. The engine destroys an S-function block under the following circumstances.

1. A simulation ends either normally or as a result of invoking `ssSetErrorStatus`.
2. A user deletes the block.
3. The engine eliminates the block as part of a block reduction optimization (see "Block reduction").

If this option is not set, the engine calls your S-function's `mdlTerminate` method only if the `mdlStart` method of at least one block in the model containing the S-function executed without error.

Example See the S-function `sfun_runtime3.c` for an example.

See Also `mdlTerminate`

SS_OPTION_CAN_BE_CALLED_CONDITIONALLY

Purpose Specify this S-function can be called conditionally

Description Specifies that the S-function can be called conditionally by other blocks. The Simulink engine uses this option to determine if the S-Function block can be moved into the execution context of the conditionally executed subsystem in which the S-function resides. See "Conditional Execution Behavior" in Using Simulink for more information.

Example See the S-function `sdotproduct.c` used in the Simulink model `sfcndemo_sdotproduct.mdl` for an example.

SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME

Purpose

Disallow constant sample time inheritance

Description

Prohibits the S-Function block that references this S-function from inheriting a constant sample time. The `SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME` option applies only to S-functions that use block-based sample times.

Note If the S-function declares the number of sample times as `PORT_BASED_SAMPLE_TIMES` it will not inherit a constant sample time unless it specifies the `SS_OPTION_ALLOW_CONSTANT_PORT_SAMPLE_TIME` option.

If the S-function specifies this option and inherits a constant sample time, i.e., a sample time of `inf`, the Real-Time Workshop product determines how to generate code for the block based on if the block is invariant.

A block is invariant if all of its ports' signals are invariant. A signal is invariant if it has a constant value during the entire simulation. A constant block sample time does not guarantee all the ports' signals are invariant. See "Inlining Invariant Signals" in the "Real-Time Workshop User's Guide" for more information.

If the block is not invariant, the Real-Time Workshop product generates code only in the `model_initialize` function. If the block is invariant, the Real-Time Workshop product eliminates the block's code altogether.

Example

See `sfblk_manswitch.c` for an example.

See Also

`SS_OPTION_ALLOW_CONSTANT_PORT_SAMPLE_TIME`

Purpose

Specify this S-function has discrete valued output

Description

Specifies this S-function has discrete valued outputs. With this option set, the Simulink engine does not assign algebraic variables to this S-function when it appears in an algebraic loop.

Purpose Improve performance of exception-free S-functions

Description Improves performance of S-functions that do not use `mexErrMsgTxt`, `mxCalloc`, or any other routines that can throw an exception. An S-function is not exception free if it contains any routine that, when called, has the potential of long-jumping out of a block of code and into another scope. See "Exception Free Code" on page 8-70 for more information.

Example See `vsfunc.c` for an example.

See Also `SS_OPTION_RUNTIME_EXCEPTION_FREE_CODE`

SS_OPTION_FORCE_NONINLINED_FCNCALL

Purpose Specify generated code format for function-call subsystems called by this S-function

Description Indicates that the Real-Time Workshop product should generate procedures for all function-call subsystems called by this S-function, instead of possibly inlining the subsystem's code. If an S-function sets this option, the Real-Time Workshop product ignores the `Inline` setting for the **Real-Time Workshop system code** option on the subsystem's Block Parameters dialog box. See "Creating Subsystems" in the "Real-Time Workshop User's Guide" for more information.

Purpose

Enable the Simulink engine to remove unnecessary S-Function blocks

Description

Specifies this S-function has no side effects. Setting this option enables the Simulink engine to remove the S-Function block referencing this S-function during dead branch elimination, if it is not needed.

Example

See the S-function `sdotproduct.c` used in the Simulink model `sfcndemo_sdotproduct.mdl` for an example.

Purpose

Specify this S-function should be placed as soon as possible

Description

Specifies that this S-function should be placed as soon as possible in the block sorted order (See "What Is Sorted Order?" in Using Simulink for more information). The Simulink engine places an S-function block using this option as far up in the sorted order as possible without changing the model's semantics. If the S-function's **Priority** block property is set, and other blocks in the model have the same priority, the engine places S-functions with this option before the other blocks with the same priority. This option is typically used by devices connecting to hardware when you want to ensure the hardware connection is completed first.

Purpose

Specify this S-function uses port-based sample times

Description

Indicates the S-function registers 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. If an S-function uses this option, it cannot inherit its sample times. See “Hybrid Block-Based and Port-Based Sample Times” on page 8-44 for more information.

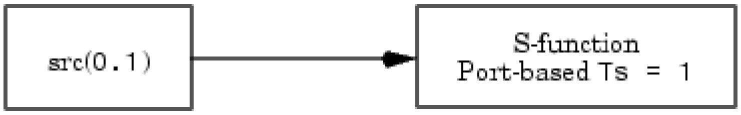
Example

See `mixedm.c` for an example.

SS_OPTION_REQ_INPUT_SAMPLE_TIME_MATCH

Purpose Specify sample times of input signal and port must match

Description Specifies that the input signal sample times must match the sample time assigned to the block input port. For example:



generates an error if this option is set. The Simulink engine does not generate an error if the block or input port sample time is inherited.

SS_OPTION_RUNTIME_EXCEPTION_FREE_CODE

Purpose Improve performance of run-time exception-free S-functions

Description Improves performance of S-functions that do not use `mexErrMsgTxt`, `mxCalloc`, or any other routines that can throw an exception inside of a run-time routines. Applicable run-time routines include `mdlGetTimeOfNextVarHit`, `mdlOutputs`, `mdlUpdate`, and `mdlDerivatives`.

See Also `SS_OPTION_EXCEPTION_FREE_CODE`

SS_OPTION_SIM_VIEWING_DEVICE

Purpose Indicate S-Function block is a `SimViewingDevice`

Description Indicates the S-Function block referencing this S-function is a `SimViewingDevice`. As long as the block meets the other requirements for a `SimViewingDevice`, i.e., no states, no outputs, etc., the Simulink engine considers the block to be an external mode block. As an external mode block, the block appears in the external mode user interface and the Real-Time Workshop product does not generate code for it. During an external mode simulation, the engine runs the block only on the host. See "Sim Viewing Devices in External Mode" on page 8-65 in Writing S-Functions for more information.

SS_OPTION_SFUNCTION_INLINED_FOR_RTW

Purpose

Specify use of TLC file during code generation

Description

Indicates the S-function has an associated TLC file and does not contain an `mdlRTW` method. Setting this option has no effect if the S-function contains an `mdlRTW` method. During code generation, if `SS_OPTION_SFUNCTION_INLINED_FOR_RTW` is set and the Real-Time Workshop product cannot locate the S-function's TLC file, the Real-Time Workshop product generates an error. If `SS_OPTION_SFUNCTION_INLINED_FOR_RTW` is not set but the Real-Time Workshop product does locate a TLC file for the S-function, it uses the TLC file.

SS_OPTION_SUPPORTS_ALIAS_DATA_TYPES

Purpose

Support data type aliases

Description

Specifies how the S-function handles signals whose data types are aliases (see `Simulink.Aliastype` for more information about data type aliases). If this option is set and the S-function's inputs and outputs use data type aliases, SimStruct macros such as `ssGetInputPortDataType` and `ssGetOutputPortDataType` return the data type IDs of those aliases. However, if this option is not set, the SimStruct macros return the data type IDs associated with the equivalent built-in data types instead. For a list of built-in values for the data type ID, see `ssGetInputPortDataType`.

Note If this option is not set and the S-function's inputs use data type aliases, the Simulink engine attempts to propagate the aliases to the S-function's outputs. However, this process can fail, in which case the engine propagates the equivalent built-in data types instead. To explicitly control the propagation of data type aliases through an S-function, enable the `SS_OPTION_SUPPORTS_ALIAS_DATA_TYPES` option.

SS_OPTION_USE_TLC_WITH_ACCELERATOR

Purpose

Use TLC file when simulating in accelerated mode

Description

Forces the Simulink Accelerator mode to use the Target Language Compiler (TLC) inlining code for the S-function, which speeds up execution of the S-function. If this option is not set, the Simulink Accelerator mode uses the MEX version of the S-function even if 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 `mdlStart` or `mdlInitializeConditions` functions together with the TLC `Outputs`, `Update`, or `Derivatives` functions. Also, this option indicates that the TLC inlining code should be used when generating a simulation target for a reference submodel that contains this S-function.

Note The Simulink Accelerator mode does not require the Real-Time Workshop product to run an inlined S-function. However, to ensure that the inlined S-function can run in accelerated mode in current and future Simulink releases, the TLC file for the S-function must use documented TLC functions to access the `CompiledModel` structure.

Example

See the S-function `timestwo.c` used in the Simulink model `sfcndemo_timestwo.mdl` for an example.

Purpose Specify this S-function supports code reuse

Description Signifies that this S-function is compatible with the Real-Time Workshop product subsystem code reuse feature. See “Writing S-Functions That Support Code Reuse” in the "Real-Time Workshop User’s Guide" for more information. If this option is not set, the Real-Time Workshop product will not reuse any subsystem containing this S-Function.

Example See `timestwo.c` for an example.

Examples

Use this list to find examples in the documentation.

S-Function Features

S-Function Examples

Writing S-Functions in M

S-Function Builder

Writing S-Functions in C

Creating Fortran S-Functions

Using Work Vectors

Index