SIMULINK®

Dynamic System Simulation for MATLAB®

Modeling

Simulation

Implementation

The MATH WORKS Inc.

Writing S-Functions

Version 3

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 Mail

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

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

Writing S-Functions

Printing History: March 1998 Beta version for Release 11
October 1998 First printing Revised for Simulink 3.0 (Release 11)

Contents

1 Overview of S-Functions

2 Writing S-Functions As M-Files

3 Writing S-Functions As C-MEX files

4 Guidelines for Writing C MEX S-Functions

1

Overview of S-Functions

Introduction

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

S-functions allow you to add your own algorithms to Simulink models. You can write your algorithms in MATLAB® or C. By following a set of simple rules, you can implement your algorithms in an S-function. After you have written your S-function and placed its name in an S-Function block (available in the Nonlinear Block sublibrary), you can customize the user interface by using masking.

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

What Is an S-Function?

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

S-functions use a special calling syntax that enables you to interact with Simulink's equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks.

The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems. As a result, nearly all Simulink models can be described as S-functions.

S-functions are incorporated into Simulink models by using the S-Function block in the Nonlinear Block sublibrary. Use the S-Function block's dialog box to specify the name of the underlying S-function, as illustrated in the figure below:

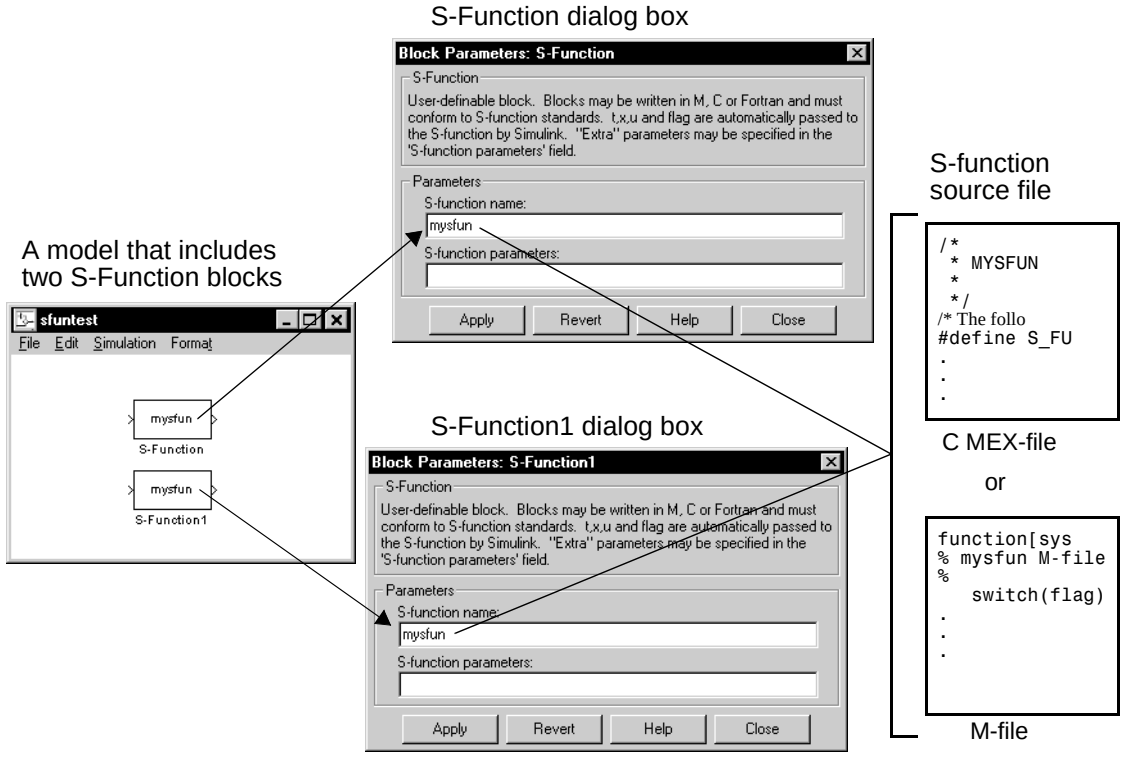


Figure 1-1: The Relationship Between an S-Function Block, Its Dialog Box, and the Source File That Defines the Block's Behavior

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

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

When to Use an S-Function

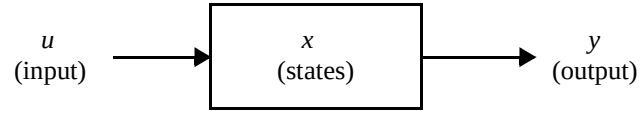
The most common use of S-functions is to create custom Simulink blocks. You can use S-functions for a variety of applications, including:

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

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

How S-Functions Work

Each block within a Simulink model has the following general characteristics: a vector of inputs, *u*, a vector of outputs, *y*, and a vector of states, *x*, as shown by this illustration:



The state vector may consist of continuous states, discrete states, or a combination of both. The mathematical relationships between the inputs, outputs, and the states are expressed by the following equations:

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

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

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

where $x = x_c + x_d$

In M-file S-functions, Simulink partitions the state vector into two parts: the continuous states and the discrete states. The continuous states occupy the first part of the state vector, and the discrete states occupy the second part. For

blocks with no states, x is an empty vector. In MEX-file S-functions, there are two separate state vectors for the continuous and discrete states.

Simulation Stages and S-Function Routines

Simulink makes repeated calls during specific stages of simulation to each block in the model, directing it to perform tasks such as computing its outputs, updating its discrete states, or computing its derivatives. Additional calls are made at the beginning and end of a simulation to perform initialization and termination tasks.

The figure below illustrates how Simulink performs a simulation. First, Simulink initializes the model; this includes initializing each block, including S-functions. Then Simulink enters the *simulation loop*, where each pass through the loop is referred to as a *simulation step*. During each simulation step, Simulink executes your S-function block. This continues until the simulation is complete:

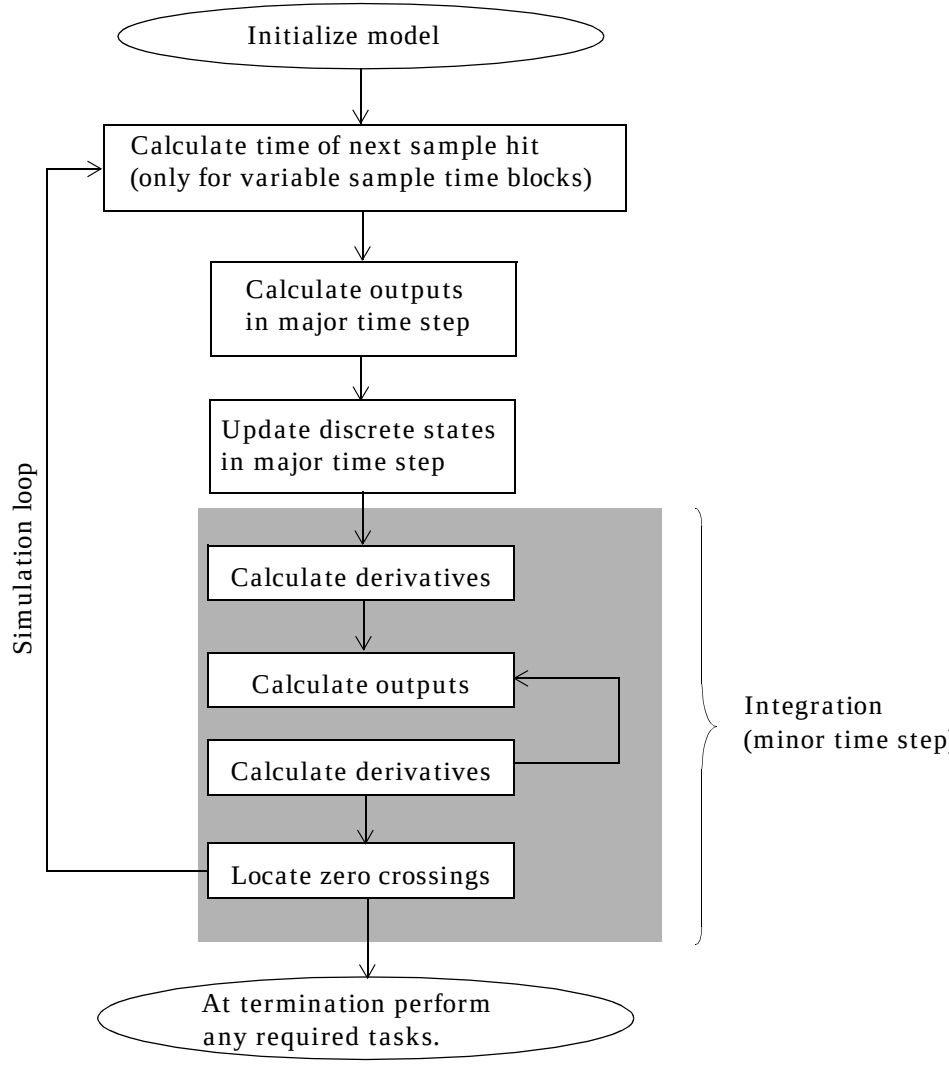


Figure 1-2: How Simulink Performs Simulation

Simulink makes repeated calls to S-functions in your model. During these calls, Simulink calls *S-function routines (*also called *methods),* which perform tasks required at each stage. These tasks include:

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

Overview of M-File and C MEX S-Functions

In M-file S-functions, the S-function routines are implemented as M-file subfunctions. In C MEX S-functions, they are implemented as C functions. All the S-function routines available to M-file S-functions exist for C MEX S-functions as well. However, Simulink provides a larger set of S-function routines for C MEX S-functions. If an S-function routine exists for both M-file and C MEX S-functions, its name is the same for both.

For an M-file S-function, Simulink passes a `flag` parameter to the S-function. The `flag` indicates the current simulation stage. You must write M-code that

calls the appropriate functions for each `flag` value. For a C MEX S-function, Simulink calls the S-function routines directly. This table lists the simulation stages, the corresponding S-function routines, and the associated `flag` value for M-file S-functions.

Table 1-1: Simulation Stages

Simulation Stage	S-Function Routine	Flag (M-File S-Functions)
Initialization	`mdlInitializeSizes`	`flag = 0`
Calculation of next sample hit (optional)	`mdlGetTimeOfNextVarHit`	`flag = 4`
Calculation of outputs	`mdlOutputs`	`flag = 3`
Update discrete states	`mdlUpdate`	`flag = 2`
Calculation of derivatives	`mdlDerivatives`	`flag = 1`
End of simulation tasks	`mdlTerminate`	`flag = 9`

C MEX S-function routines must have exactly the names shown in Table 1-1. In M-file S-functions, you must provide code that, based on the `flag` value, calls the appropriate S-function routine. A template M-file S-function, `sfuntmpl.m`, is located in *`matlabroot`*`/toolbox/simulink/blocks`. This template uses a `switch` statement to handle the `flag` values. All you have to do is place your code in the correct S-function routine.

In C MEX S-functions, Simulink directly calls the correct S-function routine for the current simulation stage. A template S-function written in C called `sfuntmpl.c`, located under `simulink/src`, is supplied with Simulink. For a more amply commented version of the template, see `sfuntmpl.doc` in the same directory.

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

S-Function Concepts

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

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

Direct Feedthrough

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

- The output function (`mdlOutputs` or `flag==3`) is a function of the input `u`. That is, there is direct feedthrough if the input `u` is used in equations defined in `mdlOutputs`. Outputs may also include graphical outputs, as in the case of an XY Graph scope.
- It is a variable sample time S-function (calls `mdlGetTimeOfNextVarHit` or `flag==4`) and the computation of the next sample hit requires the input `u`.

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

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

Outputs: $y = x$

Derivative: $\dot{x} = u$

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

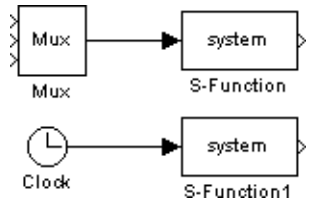
Dynamically Sized Inputs

S-functions can be written to support arbitrary input widths. In this case, the actual input width is determined dynamically when a simulation is started by evaluating the width of the input vector driving the S-function. The input

width can also be used to determine the number of continuous states, the number of discrete states, and the number of outputs.

Within an M-file S-function, to indicate that the input width is dynamically sized, specify a value of -1 for the appropriate fields in the `sizes` structure, which is returned during the `mdlInitializeSizes` call. You can determine the actual input width when your S-function is called by using `length(u)`. If you specify a width of 0, then the input port will be removed from the S-function block. Handling of port widths is similar for C MEX S-function. C MEX S-functions also let you have multiple input/output ports.

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



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

C MEX S-functions give you more flexibility in specifying the widths of input and output ports. See “Creating More Complex C MEX S-Functions” on page 3–10.

Setting Sample Times and Offsets

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

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

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

  ```
  TimeHit = (n * period) + offset
  ```

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

  If you define a discrete sample time, Simulink calls the S-function `mdlOutput` and `mdlUpdate` routines at each sample time hit (as defined in the above equation).
- Variable sample time — A discrete sample time where the intervals between sample hits can vary. At the start of each simulation step, S-functions with variable sample times are integrated for the time of next hit.
- Inherited sample time — Sometimes an S-Function block has no inherent sample time characteristics (that is, it is either continuous or discrete, depending on the sample time of some other block in the system). You can specify that the block's sample time is *inherited*. A simple example of this is a Gain block that inherits its sample time from the block driving it.

  A block can inherit its sample time from:

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

  To set a block's sample time as inherited, use -1 as the sample time. For more information on the propagation of sample times, see "Sample Time Colors" in *Using Simulink*.

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

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

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

where

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

and the italics indicate a real value is required.

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

```
[INHERITED_SAMPLE_TIME, 0.0]
```

or

```
[INHERITED_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]
```

where

```
INHERITED_SAMPLE_TIME = -1.0
```

The following guidelines may help you specify sample times:

- A continuous S-function that changes during minor integration steps should register the [CONTINUOUS_SAMPLE_TIME, 0.0] sample time.
- A continuous S-function that does not change during minor integration steps should register the [CONTINUOUS_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET] sample time.

- A discrete S-function that changes at a specified rate should register the discrete sample time pair, [*discrete_sample_time_period*, *offset*], where

  $discrete\_sample\_period > 0.0$

  and

  $0.0 \leq offset < discrete\_sample\_period$

- A discrete S-function that changes at a variable rate should register the variable step discrete sample time:

  `[VARIABLE_SAMPLE_TIME, 0.0]`

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

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

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

  For example, most discrete-time S-functions must register this sample time when inheriting the input sample time.

Sample S-Functions

It may be helpful to examine some sample S-functions as you read the next chapters. Examples are stored in these subdirectories under the MATLAB root directory:

- M-files: `toolbox/simulink/blocks`
- C MEX-files: `simulink/src`

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

Table 1-2: Example M-File S-Functions

Filename	Description
csfunc.m	Defines a continuous system in state-space format.
dsfunc.m	Defines a discrete system in state-space format.
vsfunc.m	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.
mixed.m	Implements a hybrid system consisting of a continuous integrator in series with a unit delay.
vdpm.m	Implements the Van der Pol equation.
simom.m	An example state-space M-file S-function with internal A, B, C, and D matrices. This S-function implements dx/at = Ax + By y = Cx + Du where x is the state vector, u is the input vector, and y is the output vector. The A, B, C, and D matrices are embedded in the M-file.
simom2.m	An example state-space M-file S-function with external A, B, C, and D matrices. The state-space structure is the same as in simom.m, but the A, B, C, and D matrices are provided externally as parameters to this file.
limintm.m	Implements a continuous limited integrator where the output is bounded by lower and upper bounds and includes initial conditions.

Table 1-2: Example M-File S-Functions (Continued)

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

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

Table 1-3: Example C MEX S-Functions

Filename	Description
`timestwo.c`	A basic C MEX S-function that doubles its input.
`csfunc.c`	An example C MEX S-function for defining a continuous system.
`dsfunc.c`	An example C MEX S-function for defining a discrete system.
`dlimint.c`	Implements a discrete-time limited integrator.
`vsfunc.c`	Illustrates how to create a variable step block in Simulink. This block implements a variable step delay in which the first input is delayed by an amount of time determined by the second input.
`mixed.c`	Implements a hybrid dynamical system consisting of a continuous integrator (1/s) in series with a unit delay (1/z).

Table 1-3: Example C MEX S-Functions (Continued)

Filename	Description
`mixedmex.c`	Implements a hybrid dynamical system with a single output and two inputs.
`quantize.c`	An example MEX-file for a vectorized quantizer block. Quantizes the input into steps as specified by the quantization interval parameter, `q`.
`resetint.c`	A reset integrator.
`sftable2.c`	A two-dimensional table lookup in S-function form.
`sfun_dynsize.c`	A simple example of how to size outputs of an S-function dynamically.
`sfun_errhndl.c`	A simple example of how to check parameters using the `mdlCheckParams` S-function routine.
`sfun_fcncall.c`	An example of an S-function that is configured to execute function-call subsystems on the first and third output element.
`sfun_multiport.c`	An example of an S-function that has multiple input and output ports.
`sfun_multirate.c`	An example of an S-function that demonstrates how to specify port-based sample times.
`sfun_zc.c`	An example of an S-function that has nonsampled zero crossings to implement `abs(u)`. This S-function is designed to be used with a variable step solver.
`sfun_zc_sat.c`	Saturation example that uses zero crossings.
`sfunmem.c`	A one integration-step delay and hold “memory” function.
`vdpm.c`	Implements the van der Pol equation.

Table 1-3: Example C MEX S-Functions (Continued)

Filename	Description

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

`stspace.c`	Implements a set of state-space equations. You can turn this into a new block by using the S-Function block and mask facility. This example MEX-file performs the same function as the built-in State-Space block. This is an example of a MEX-file where the number of inputs, outputs, and states is dependent on the parameters passed in from the workspace. Use this as a template for other MEX-file systems.
`stvctf.c`	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.
`stvdct.f`	Implements 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.
`limintc.c`	Implements a limited integrator.
`vdlmint.c`	Implements a discrete-time vectorized limited integrator.
`vlimint.c`	Implements a vectorized limited integrator.

2

Writing S-Functions As M-Files

Introduction

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

Simulink provides a template M-file S-function that includes statements that define necessary functions, as well as comments to help you write the code needed for your S-function block. This template file, `sfuntmpl.m`, is in the directory `toolbox/simulink/blocks` under the MATLAB root directory.

M-file S-functions work by making a sequence of calls to S-function routines, which are M-code functions that perform tasks required by your S-function. This table lists the S-function routines available to M-file S-functions.

Table 2-1: *M-File S-Function Routines*

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

Building S-functions can be thought of as two separate tasks:

- Initializing block characteristics, including number of inputs, outputs, initial conditions of continuous and discrete states, and sample times
- Placing your algorithms in the appropriate S-function routine

Defining S-Function Block Characteristics

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

To give Simulink this information, call the `simsizes` function at the beginning of `mdlInitializeSizes`:

```
sizes = simsizes;
```

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

Table 2-2: Fields in the sizes Structure

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

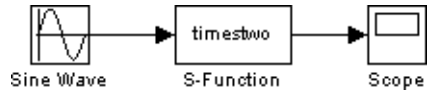
After you initialize the `sizes` structure, call `simsizes` again:

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

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

A Simple M-File S-Function Example

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



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

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

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

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

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

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

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

The first four input arguments, which Simulink passes to the S-function, must be the variables `t`, `x`, `u`, and `flag`:

- `t`, the time
- `x`, the state vector (required even if, as in this case, there are no states)
- `u`, the input vector
- `flag`, the parameter that controls the S-function subroutine calls at each simulation stage

Simulink also requires that the output parameters, `sys`, `x0`, `str`, and `ts` be placed in the order given. These parameters are:

- `sys`, a generic return argument. The values returned depend on the flag value. For example, for `flag = 3`, `sys` contains the S-function outputs.
- `x0`, the initial state values (an empty vector if there are no states in the system). `x0` is ignored, except when `flag = 0`.
- `str`, reserved for future use. M-file S-functions must set this to the empty matrix, `[]`.
- `ts`, a two column matrix containing the sample times and offsets of the block. Continuous systems have their sample time set to zero. The hybrid example, which starts on page 2-14, demonstrates an S-function with multiple sample times.

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

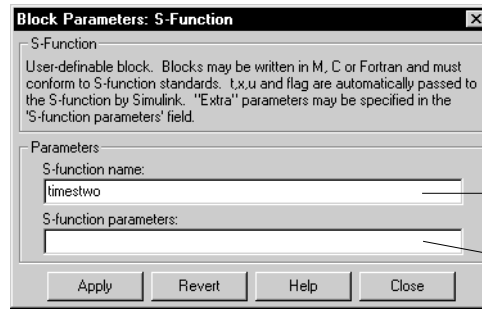
  ```
  ts = [.25 0; 1.0 .1];
  ```

  For more discussion of sample times, see “Sample Time Colors” in *Using Simulink*.

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

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

% End of mdlOutputs.
```

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



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

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

You can now run this simulation.

Examples of M-File S-Functions

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

- Continuous
- Discrete
- Hybrid
- Variable-step

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

Example - Continuous State S-Function

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

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

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

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

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

  case { 2, 4, 9 } % Unused flags
    sys = [];
  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end
% End of csfunc.
```

```
%=============================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%=============================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(A,B,C,D)
%
% Call simsizes for a sizes structure, fill it in and convert it
% to a sizes array.
%
sizes = simsizes;
sizes.NumContStates  = 2;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = 2;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1;     % Matrix D is nonempty.
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
%
% Initialize the initial conditions.
%
x0 = zeros(2,1);
%
% str is an empty matrix.
%
str = [];
%
% Initialize the array of sample times; in this example the sample
% time is continuous, so set ts to 0 and its offset to 0.
%
ts = [0 0];
% End of mdlInitializeSizes.
%
%=============================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================
function sys = mdlDerivatives(t,x,u,A,B,C,D)
sys = A*x + B*u;
```

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

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

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

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

Each time the `mdlDerivatives` routine is called it must explicitly set the value of all derivatives. The derivative vector does not maintain the values from the last call to this routine. The memory allocated to the derivative vector changes during execution.

Example - Discrete State S-Function

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

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

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

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

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

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

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

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

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

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

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

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

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

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

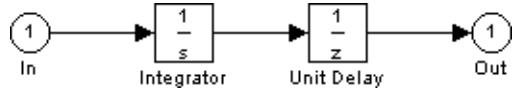
```
x(n+1) = Ax(n) + Bu(n)
y(n)   = Cx(n) + Du(n)
```

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

Example - Hybrid System S-Functions

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

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



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

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

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

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

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

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

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

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

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

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

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

% End of mdlOutputs.
```

Example - Variable Step S-Functions

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

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

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

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

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

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

  case { 1, 9 }
    sys = []; % Unused flags
  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end
% End of vsfunc.
%=============================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%=============================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes
%
% Call simsizes for a sizes structure, fill it in and convert it
% to a sizes array.
%
```

```
sizes = simsizes;
sizes.NumContStates  = 0;
sizes.NumDiscStates  = 1;
sizes.NumOutputs     = 1;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1;  % flag=4 requires direct feedthrough
                           % if input u is involved in
                           % calculating the next sample time
                           % hit.
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
%
% Initialize the initial conditions.
%
x0 = [0];
%
% Set str to an empty matrix.
%
str = [];
%
% Initialize the array of sample times.
%
ts = [-2 0];      % variable sample time
% End of mdlInitializeSizes.
%
%=============================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time
% step requirements.
%=============================================================
%
function sys = mdlUpdate(t,x,u)
sys = u(1);
% End of mdlUpdate.
%
%=============================================================
% mdlOutputs
% Return the block outputs.
%=============================================================
%
```

```
function sys = mdlOutputs(t,x,u)
sys = x(1);
% end mdlOutputs
%
%=============================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the
% result is absolute time.
%=============================================================
%
function sys = mdlGetTimeOfNextVarHit(t,x,u)
sys = t + u(2);
% End of mdlGetTimeOfNextVarHit.
```

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

Passing Additional Parameters

Simulink always passes `t`, `x`, `u`, and `flag` into S-functions. It is possible to pass additional parameters into your M-file S-function. For an example of how to do this, see `limintm.m` in the `toolbox/simulink/blocks` directory.

3

Writing S-Functions As C-MEX files

Introduction

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

C MEX-file S-functions have the same structure and perform the same functions as M-file S-functions. In addition, C MEX S-functions provide you with more functionality than M-file S-functions. Simulink includes a template file for writing C MEX S-functions, called `sfuntmpl.c` and a more complete version called `sfuntmpl.doc` (both located in `simulink/src`).

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

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

static void mdlInitializeSizes(SimStruct *S)
{
}

<additional S-function routines/code>

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

`mdlInitializeSizes` is the first routine Simulink calls when interacting with the S-function. There are several other `mdl*` routines within an S-function. All S-functions must conform to this format. This means that different S-functions can implemented using S-function routines of the same name but different

contents. After Simulink calls `mdlInitializeSizes`, it then interacts with the S-function through various other routines (all starting with `mdl`). At the end of a simulation, `mdlTerminate` is called.

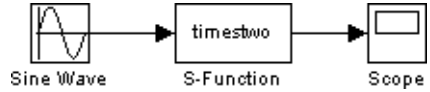
Unlike M-file S-functions, there is not an explicit `flag` parameter associated with each C MEX S-function routine. This is because Simulink automatically calls each S-function routine at the appropriate time during its interaction with the S-function. Also, there are S-function routines associated with C MEX S-functions that don't have counterparts in M-file S-functions.

Simulink maintains information about the S-function in a data structure called the `SimStruct`. The `#include` file, `simstruc.h`, that defines the `SimStruct` provides 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`. The statement that includes the definition of the `SimStruct` is shown in "Statements Required at the Top of S-Functions" on page 3–10. The macros that access the `SimStruct` are described in "The C MEX S-Function SimStruct" on page 3–97. Commonly used macros that are used to access the `SimStruct` are described in the following sections.

Writing Basic C MEX S-Functions

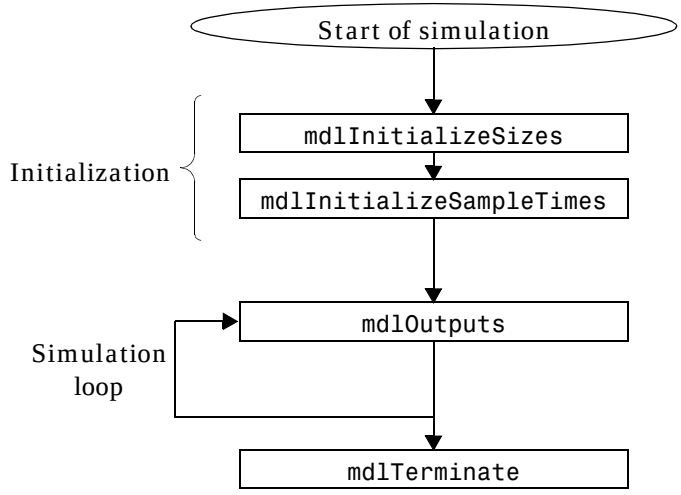
This section discusses basic C MEX S-functions, where basic means an S-function that contains only the required S-function routines. However, the contents of the routines can be as complex as you like. You have the freedom to place any logic in S-function routines as long as the routines conform to their required formats. This section presents a basic C MEX S-function by showing how to implement an S-function that takes its input and multiplies it by two (see *matlabroot*`/simulink/src/timestwo.c`).

In the following model, the `timestwo` S-function is used to double the amplitude of a sine wave and plot it in a scope:



Within the block dialog for the S-function, `timestwo` has been specified as the name; the parameters field is empty.

The `timestwo` S-function contains the S-function routines shown in this figure:



To incorporate this S-function into Simulink, create a source file called `timestwo.c` and place the S-function routines of the above figure in it. Typing

```
mex timestwo.c
```

at the command line directs MATLAB to compile and link the `timestwo.c` file. This creates a dynamically loadable executable for Simulink's use.

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

This table shows the descriptions of the S-function routines required in the `timestwo` S-function.

Table 3-1: S-Function Routines Called During Simulation of the timestwo Example

S-Function Routine	Description
`mdlInitializeSizes`	Simulink calls this routine while editing the model to determine the number of input and output ports. Simulink also calls it at the start of simulation to inquire about the sizes of the ports and any other objects (such as the number of states) needed by the S-function.
`mdlInitializeSampleTimes`	Simulink calls this routine to set the sample time(s) of the S-function. In this example, it executes whenever the driving block executes. Therefore, it has a single inherited sample time, `SAMPLE_TIME_INHERITED`.

Table 3-1: S-Function Routines Called During Simulation of the timestwo Example (Continued)

S-Function Routine	Description
`mdlOutputs`	Calculation of outputs. For our `timestwo` S-function, `mdlOutputs` takes the input, multiplies it by two, and writes the answer to the output. This routine is called during the simulation loop each time the output of `timestwo` needs to be updated. For the above block diagram, it is called at every time step.
`mdlTerminate`	Perform tasks at end of simulation. The `timestwo` S-function has no termination tasks to perform; therefore, this routine is empty.

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

```
#define S_FUNCTION_NAME  timestwo
#define S_FUNCTION_LEVEL 2

#include “simstruc.h”

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

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

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

    ssSetNumSampleTimes(S, 1);

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

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

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

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

static void mdlTerminate(SimStruct *S){}

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

There is more information that the `timestwo.c` example requires:

- Defines and includes — The example specifies the name of the S-function (`timestwo`) and that the S-function is in the *level 2* format (for more information about level 1 and level 2 S-functions, see "Converting Level 1 C MEX S-Functions to Level 2" on page 3-118). After defining these two items, the example includes `simstruc.h`, which is a header file that gives access to the `SimStruct` data structure and the MATLAB Application Program Interface (API) functions:

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

- `mdlInitializeSizes` specifies the following for the `timestwo` example:
  - Zero parameters—This means that the **S-function parameters** field of the S-functions's dialog box must be empty. If it contains any parameters, Simulink will report a parameter mismatch.
  - One input port and one output port—The widths of the input and output port are dynamically sized. This tells Simulink to multiply each element of the input signal to the S-function by two and to place the result in the output signal. Note that the default handling for dynamically sized S-functions for this case (one input and one output) is that the input and output widths are equal.
  - One sample time. You must specify the actual value of the sample time in the `mdlInitializeSampleTimes` routine.
  - The code is exception free. Specifying exception free code speeds up execution of your S-function. Care must be taken when specifying this option. In general, if your S-function isn't interacting with MATLAB, it is safe to specify this option. For more details, see "Error Handling" on page 3–12.
- `mdlInitializeSampleTimes:`
  - The sample time is inherited from the driving block. This means that the S-function will run whenever it receives input from the block that is connected to the S-function block's input port.

- `mdlOutput`:
  - The numerical calculation. `mdlOutputs` tells Simulink to multiply the input signal by 2.0 and place the result in the output signal.
  - To access the input signal, use:

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

    where `uPtrs` is a vector of pointers and *must* be accessed using:

    ```
    *uPtrs[i]
    ```

    For more details, see "Data View of S-Functions" on page 3–19.
  - To access the output signal, use

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

    This returns a contiguous array for the block's outputs.
  - The S-function loops over the width of the signal passing through the block. To access the width, you can check the input or output port width. Here the output port width was evaluated.
- `mdlTerminate`:
  - This is a mandatory S-function routine. However, the `timestwo` S-function doesn't need to perform any termination actions, so this routine is empty.
- At the end of the S-function, specify code that attaches this example to either Simulink or the Real-Time Workshop:

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

Creating More Complex C MEX S-Functions

There are numerous S-function routines available for use in C MEX S-functions. Simulink provides template C MEX-file S-functions that include statements that define the necessary functions, as well as comments that should help you write the code needed for your S-function block. The template file, `sfuntmpl.c`, which can be found in the directory `simulink/src` below the MATLAB root directory, contains commonly used S-function routines. A template containing all available routines (as well as more comments) can be found in `sfuntmpl.doc` in the same directory.

Note We recommend that you use the C MEX-file template when developing MEX S-functions. See section "Converting Level 1 C MEX S-Functions to Level 2" on page 3–118.

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 SFUNCTION_LEVEL 2
#include "simstruc.h"
```

Where *`your_sfunction_name_here`* is the name of your S-function (i.e., what you enter in the Simulink S-Function block dialog). These statements give you access to the `SimStruct` data structure that contains pointers to the data used by the simulation. The included code also defines the macros used to store and retrieve data in the `SimStruct`, described in detail in "The C MEX S-Function SimStruct" on page 3–97. In addition, the code specifies that you are using the level 2 format of S-functions.

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

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

Table 3-2: Header Files Included by Simstruc.h When Compiling as a MEX-File

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

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

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

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

Statements Required at the Bottom of S-Functions

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

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

These statements select the appropriate code for your particular application.

- `simulink.c` is included if the file is being compiled into a MEX-file.
- `cg_sfun.h` is included if the file is being used in conjunction with the Real-Time Workshop to produce a stand-alone or real-time executable.

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

Conditionally Compiling S-Functions

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

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

Error Handling

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

In the `timestwo` example above, there was no need for significant error handling. The only piece of code that performed error handling was the early `return` in `mdlInitializeSizes`. Here Simulink issues a parameter mismatch if you entered parameters in the dialog box. If your S-function has parameters whose contents you need to validate, use the following technique to report errors encountered:

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

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

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

The ssSetErrorStatus error handling approach is the suggested alternative to using the mexErrMsgTxt function. The function mexErrMsgTxt uses exception handling to immediately terminate S-function execution and return control to Simulink. In order to support exception handling inside of S-functions, Simulink must set up exception handlers prior to each S-function invocation. This introduces overhead into simulation.

Exception Free Code

You can avoid this overhead by ensuring that your S-function contains entirely *exception free code.* Exception free code refers to code that never long jumps. Your S-function is not exception free if it contains any routine that, when called, has the potential of long jumping. For example mexErrMsgTxt throws an exception (i.e., long jumps) when called, thus ending execution of your S-function. Using mxCalloc may cause unpredictable results in the event of a memory allocation error since mxCalloc will long jump. 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, then use the SS_OPTION_EXCEPTION_FREE_CODE S-function option. This is done by issuing the following command in the mdlInitializeSizes function:

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

Setting this option will increase the performance of your S-function by allowing Simulink to bypass the exception handling setup that is usually performed prior to each S-function invocation. Extreme care must be taken 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 will occur.

All `mex*` routines have the potential of long jumping. In addition several `mx*` routines 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 will never throw an exception: `mxGetPr`, `mxGetData`, `mxGetNumberOfDimensions`, `mxGetM`, `mxGetN`, and `mxGetNumberOfElements`.

Code in *run-time routines* can also throw exceptions. Run-time routines refer to certain S-function routines that Simulink calls during the simulation loop (see "The Calling Sequence for S-Functions" on page 3-16). The run-time routines include:

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

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

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

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

ssSetErrorStatus Termination Criteria

When you call `ssSetErrorStatus` and return from your S-function, Simulink stops the simulation and posts the error. To determine how the simulation shuts down, refer to the flow chart figure on page 3-16. If `ssSetErrorStatus` is called prior to `mdlStart`, no other S-function routine will be called. If `ssSetErrorStatus` is called in `mdlStart` or later, `mdlTerminate` will be called.

Overview of the C MEX S-Function Routines

The following figure shows the calling structure, including optional S-function routines (methods), of a C MEX S-function:

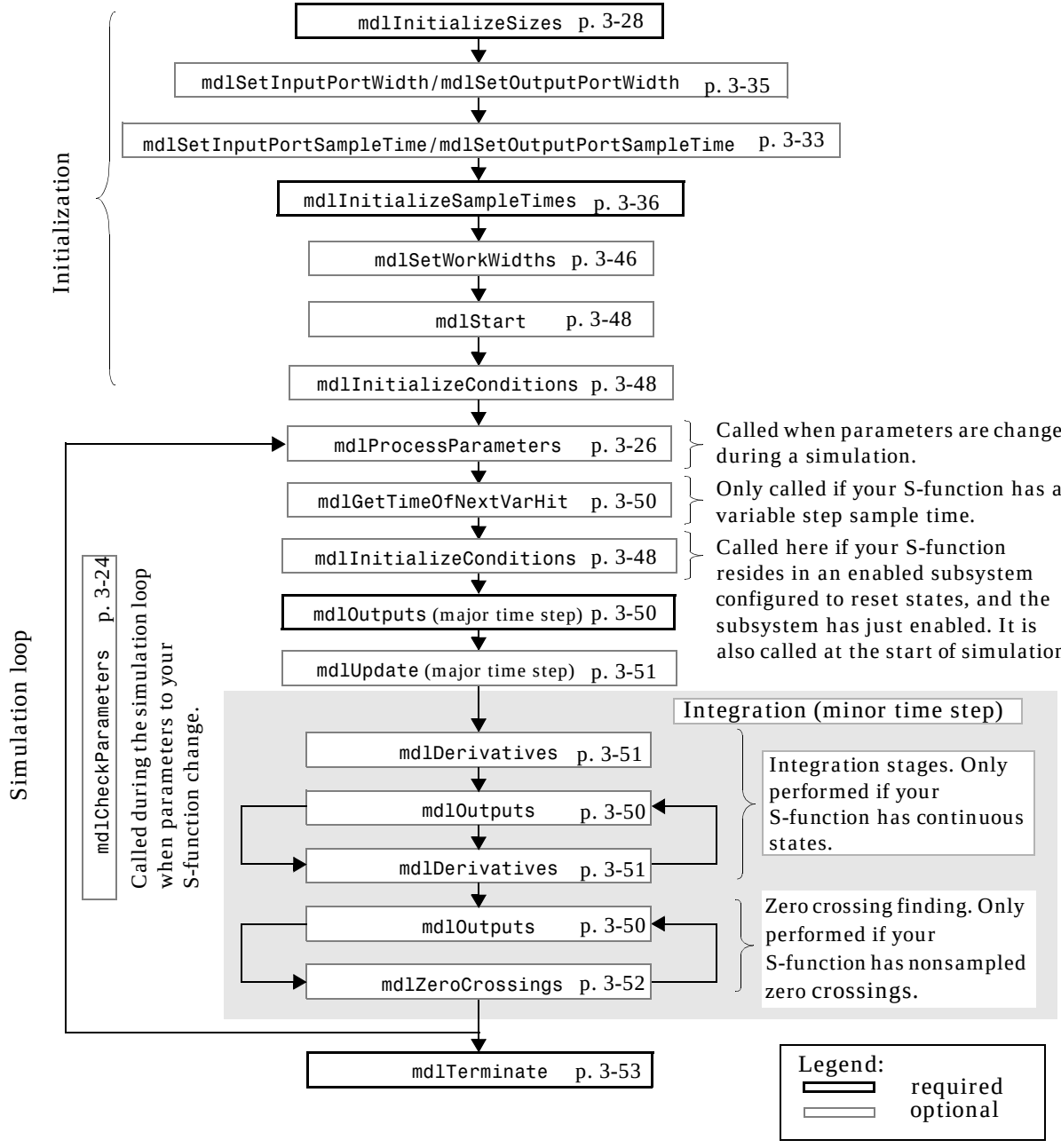


Figure 3-1: The Calling Sequence for S-Functions

The following sections discuss each of these routines and give an overview of the more common macros used to access the `SimStruct` data structure. A full list of the macros is given in "The C MEX S-Function SimStruct" on page 3–97.

Alternate Calling Structure for the Real Time Workshop

If you use the Real-Time Workshop to generate code for a model that contains S-functions, Simulink does not go through the entire calling sequence outlined above. Initialization proceeds as outlined above until Simulink reaches the `mdlInitializeConditions` routine. After calling `mdlInitializeConditions`, Simulink calls `mdlRTW`, an S-function routine unique to the Real-Time Workshop, `mdlTerminate`, and exits. This picture shows the calling sequence:

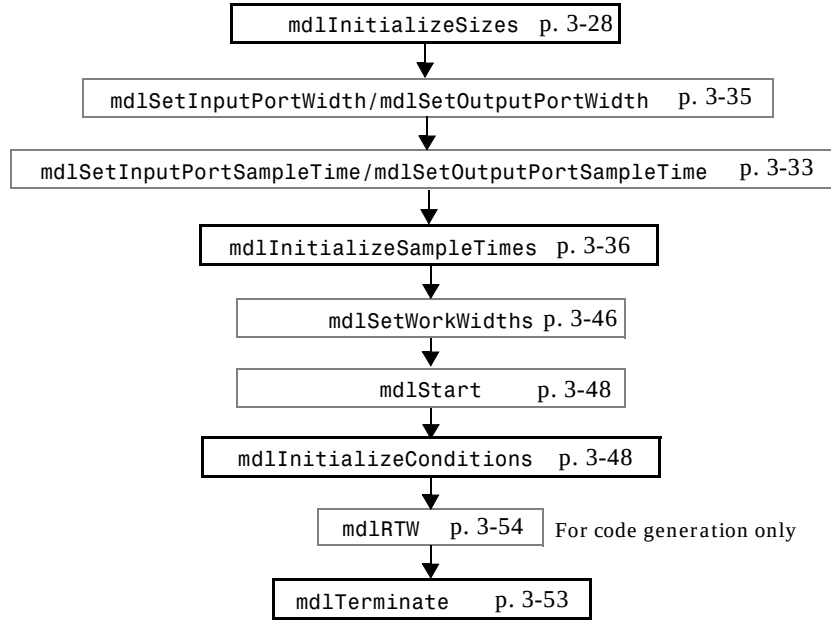


Figure 3-2: S-Function Calling Sequence when Simulink is Used with the Real-Time Workshop

For more information about the Real-Time Workshop and how it interacts with S-functions, see *The Real-Time Workshop User's Guide* and *The Target Language Compiler Reference Guide*.

Alternate Calling Structure for External Mode

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

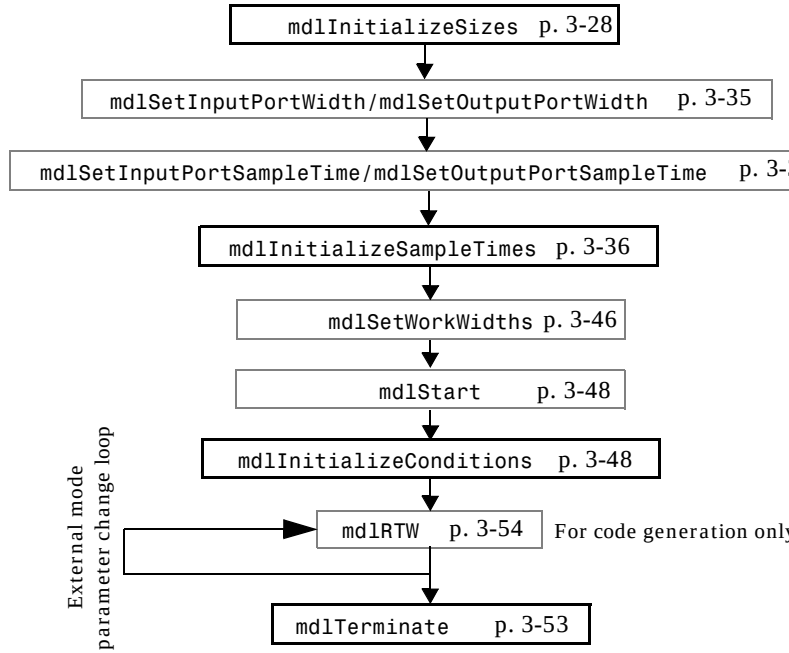


Figure 3-3: S-Function Calling Sequence when Simulink Runs in External Mode

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

Note Running Simulink in external mode requires the Real-Time Workshop. For more information about external mode, see the *Real-Time Workshop User's Guide.*

Data View of S-Functions

S-function blocks have input and output signals, parameters, 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

S-function blocks can be parameterized by passing parameters them using the S-function block dialog box.

The following picture shows the general mapping between these various types of data:

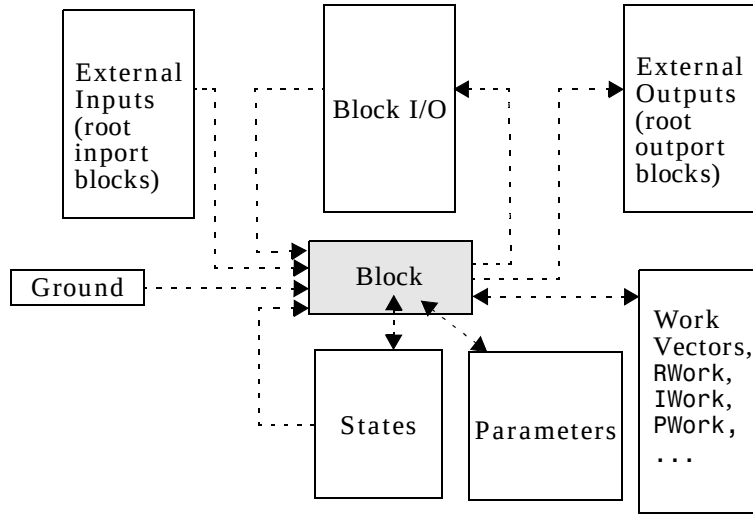


Figure 3-4: The Relationships Among S-Functions and S-Function Data

The length of the various signals and vectors is configured in the `mdlInitializeSizes` routine. The signals as well as their length can be accessed in S-function routines that are called during the simulation loop.

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

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

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

```
*uPtrs[element]
```

as described by this figure:

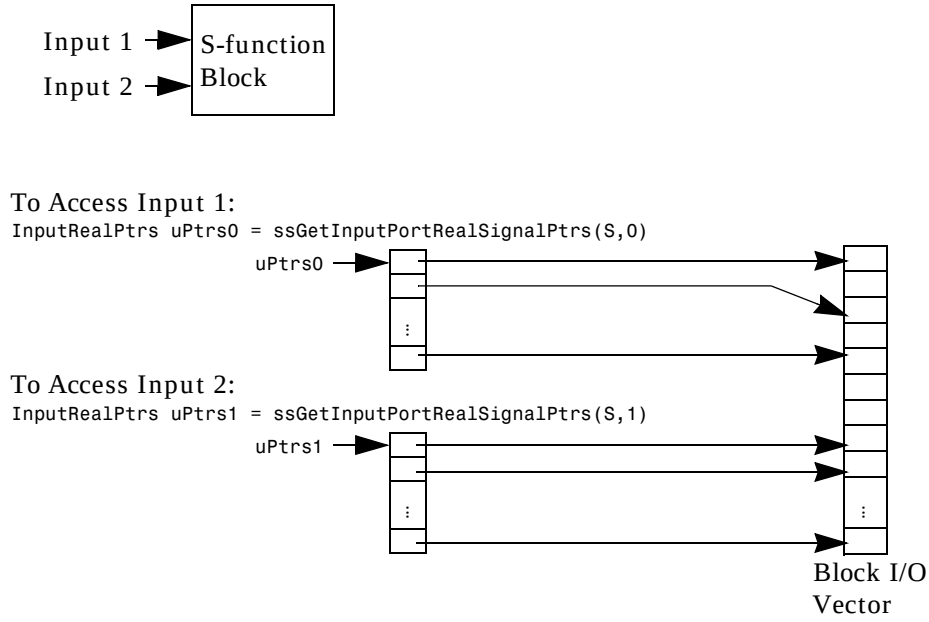


Figure 3-5: Accessing Input Data from S-Function Input Ports

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

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

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 figure above shows that the input array of pointers may 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);
InputRealPtrs uPtrs = ssGetInputPortRealSignalPtrs(S,inputPortIndex);
real_T *y = ssGetOutputPortSignal(S,outputPortIdx);

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

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

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

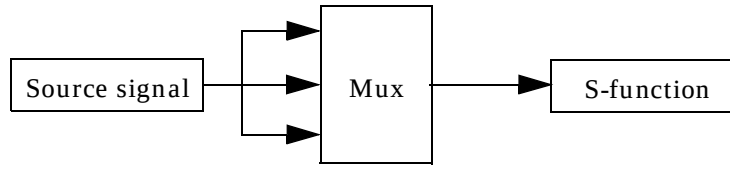
just below the initialization of `uPtrs` and replace the inner part of the above loop with

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

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

To verify that you are correctly accessing wide input signals, pass a replicated signal to each input port of your S-function. This is done by creating a Mux block with the number of input ports equal to the width of the desired signal

entering your S-function. Then the driving source should be connected to each input port as shown in this figure:



Checking and Processing S-Function Parameters

You can provide parameters to S-functions that can be changed interactively using the **S-Function parameters** field of the block's dialog box. If you define parameters interactively, follow these steps when you create the S-function:

1. Determine the order in which the parameters are to be specified in the block's dialog box.

2. In the `mdlInitializeSizes` function, use the `ssSetNumSFcnParams` macro to tell Simulink how many parameters are passed in to the S-function. Specify `S` as the first argument and the number of parameters you are defining interactively as the second argument.

3. Access these input arguments in the S-function using the `ssGetSFcnParam` macro. Specify `S` as the first argument and the relative position of the parameter in the list entered on the dialog box (0 is the first position) as the second argument.

When you run the simulation, specify parameter names or values in the **S-Function parameters** field of the block's dialog box. The order of the parameters names or values must be identical to the order that you defined them in step 1 above. If you specify variable names, they do not need to be the same as the names used in the MEX-file.

For example, the following code is part of a device driver S-function. Four input parameters are used: `BASE_ADDRESS_PRM`, `GAIN_RANGE_PRM`, `PROG_GAIN_PRM`,

and NUM_OF_CHANNELS_PRM. The code uses #define statements to associate particular input arguments with the parameter names.

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

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

The mdlInitializeSizes function contains this statement:

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

Parameter Changes

To notify your S-function of parameter changes during a Simulink simulation, the S-function must register an mdlCheckParameters routine. This S-function routine is called any time after mdlInitializeSizes has been called. If you want to process the changed parameters, register the mdlProcessParameters routine. Typically this routine caches the parameters in work vectors.

The optional S-function routines, mdlCheckParameters and mdlProcessParameters are generally used together when writing an S-function that contains parameters that can change during simulation. These routines can only be used in a C MEX S-function.

S-functions that include these routines should verify that their code robustly handles parameter changes when working with the Real-Time Workshop. When S-functions are used with the Real-Time Workshop, that is, linked with the generated code to form an executable, there is a global parameter vector that you can modify by a number of external means. One such means is external mode. When running in external mode, Simulink calls mdlCheckParameters and mdlProcessParameters in your MEX-file. After Simulink completes these calls, the external mode link changes the parameters in the generated code executable directly in memory. For more information on external mode, see *The Real-Time Workshop User's Guide*.

Typically, it is safe to have an `mdlCheckParameters` in your S-function when it is used with the Real-Time Workshop, providing that when parameter tuning is performed, it is done using Simulink external mode. If you have an `mdlProcessParameters` routine in your S-function code, you may need to inline the S-function for it to work correctly in external mode. This is necessary because only changes to the tunable parameters are passed to the generated code. For more information on inlining S-functions, see the *Target Language Compiler Reference Guide*.

The flowchart on page 3-16 shows that Simulink only calls `mdlCheckParameters` when parameters are changed during simulation. Once `mdlCheckParameters` verifies that parameter changes are correct, Simulink calls `mdlProcessParameters` to translate the parameters into a more convenient and efficient form, such as caching the values in the work vectors. In general, your S-function doesn't need to have an `mdlProcessParameters`; it is provided to help you speed up the execution of S-functions. This is achieved by only evaluating the parameters when they change.

mdlCheckParameters

Use `mdlCheckParameters` to verify that parameter settings are correct:

```
#define MDL_CHECK_PARAMETERS   /* define is required for use*/
#if defined (MDL_CHECK_PARAMETERS)
static void mdlCheckParameters(SimStruct *S)
{
}
#endif
```

`mdlCheckParameters` verifies new parameter settings whenever parameters change or are re-evaluated during a simulation.

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

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

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

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

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

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

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

mdlProcessParameters

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

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

The synopsis is:

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

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

```
#define MDL_PROCESS_PARAMETERS   /* Change to #undef to remove function */
#if defined(MDL_PROCESS_PARAMETERS) && defined(MATLAB_MEX_FILE)
static void mdlProcessParameters(SimStruct *S)
  {
    int_T  i;
    char_T *plusMinusStr;
    int_T  nInputPorts = ssGetNumInputPorts(S);
    int_T  *iwork      = ssGetIWork(S);
    if ((plusMinusStr=(char_T*)malloc(nInputPorts+1)) == NULL) {
        ssSetErrorStatus(S,"Memory allocation error in mdlStart");
        return;
    }
    if (mxGetString(SIGNS_PARAM(S),plusMinusStr,nInputPorts+1) != 0) {
        free(plusMinusStr);
        ssSetErrorStatus(S,"mxGetString error in mdlStart");
        return;
    }
    for (i = 0; i < nInputPorts; i++) {
        iwork[i] = plusMinusStr[i] == '+'? 1: -1;
    }
    free(plusMinusStr);

  }
#endif /* MDL_PROCESS_PARAMETERS */
```

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

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

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

Defining S-Function Block Characteristics

The `sizes` structure in the `SimStruct` stores essential size information about the S-Function block, including the number of inputs, outputs, states, and other block characteristics. The `sizes` structure is initialized in the

mdlInitializeSizes function. Supplied macros set values for the structure fields. If a value is not specified, it is initialized to zero.

There are additional macros that get various values. For a complete list of built-in macros that work with C language S-functions, see "The C MEX S-Function SimStruct" on page 3–97.

mdlInitializeSizes

mdlInitializeSizes is the first routine Simulink calls when interacting with an S-function. This routine specifies the sizes information Simulink uses to determine the S-function block's characteristics (number of inputs, outputs, states, etc.).

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

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

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

In addition, you can use these macros to set and verify information in your mdlInitializeSizes routine:

- Use ssGetInputPortConnected(S) and ssGetOutputPortConnected(S) macros after setting the number of input and output ports to determine if the ports are connected.
- Use ssSetSFcnParamNotTunable(S,paramIdx) when a parameter cannot change during simulation, where paramIdx starts at 0. When a parameter has been specified as "not tunable," Simulink will issue an error during

simulation (or the Real-Time Workshop external mode) if an attempt is made to change the parameter.

- If your S-function outputs are discrete (e.g., can only take on the values, 1 and 2), then specify `SS_OPTION_DISCRETE_VALUED_OUTPUT`.
- The `ssGetPath` (the Simulink full model path to the block) and `ssGetModelName` (here model refers to the S-function name) are available for use. If they are the same (that is, if `strcmp(ssGetPath(S),ssGetModelName(S))==0`), then the S-function is being executed from the MATLAB command line and is not part of a simulation.

In mdlInitializeSizes, you also must specify the number of sample times for your block. There are two ways of specifying sample times:

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

See "Setting Sample Times for C MEX S-Functions" on page 3-36 for a complete discussion of sample time issues.

The synopsis is

```
/* Function: mdlInitializeSizes ===============================================*
Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 *
 *    The direct feedthrough flag can be either 1=yes or 0=no. It should be
 *    set to 1 if the input, "u", is used in the mdlOutput function. Setting
 *    this to 0 is tells Simulink that "u" will not be used in the
 *    mdlOutput function. If you violate this, unpredictable results
 *    will occur.
 *
 *    The NumContStates, NumDiscStates, NumInputs, NumOutputs, NumRWork,
 *    NumIWork, NumPWork NumModes, and NumNonsampledZCs widths can be set to:
 *       DYNAMICALLY_SIZED    - In this case, they will be set to the actual
 *                              input width, unless you are have a
 *                              mdlSetWorkWidths to set the widths.
 *       0 or positive number - This explicitly sets item to the specified
 *                              value.
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /*
         * If the the number of expected input parameters is not equal
         * to the number of parameters entered in the dialog box return.
         * Simulink will generate an error indicating that there is a
         * parameter mismatch.
         */
        return;
    }

    ssSetNumContStates(    S, 0);   /* number of continuous states           */
    ssSetNumDiscStates(    S, 0);   /* number of discrete states             */


    /*
     * Configure the input ports. First set the number of input ports,
     * then set for each input port index starting at 0, the width
     * and wether or not the port has direct feedthrough (1=yes, 0=no).
     * The width of a port can be DYNAMICALLY_SIZED or greater than zero.
     * A port has direct feedthrough if the input is used in either
     * the mdlOutputs or mdlGetTimeOfNextVarHit functions.
     */
    if (!ssSetNumInputPorts(S, nInputPorts)) return;
    ssSetInputPortWidth(S, inputPortIdx, width);
    ssSetInputPortDirectFeedThrough(S, inputPortIdx, needsInput);

    /*
     * Configure the output ports. First set the number of output ports,
     * then set for each output port index starting at 0, the width
```

```
 * of the output port which can be DYNAMICALLY_SIZE or greater than zero.
 */
if (!ssSetNumOutputPorts(S, nOutputPorts)) return;
ssSetOutputPortWidth(S, outputPortIdx, width);

/*
 * Set the number of sample times. This must be a positive, nonzero
 * integer indicating the number of sample times or it can be
 * PORT_BASED_SAMPLE_TIMES. For multi-rate S-functions, the
 * suggested approach to setting sample times is via the port
 * based sample times routine. When you create a multirate
 * S-function, care needs to be taking to verify that when
 * slower tasks are preempted that your S-function correctly
 * manages data as to avoid race conditions. When port based
 * sample times are specified, the block cannot inherit a constant
 * sample time at any port.
 */
ssSetNumSampleTimes(   S, 1);   /* number of sample times                */

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

/*
 * Set any S-function options which must be OR'd together.
 * The available options are:
 * SS_OPTION_EXCEPTION_FREE_CODE - if your S-function does not use
 *   mexErrMsgTxt, mxCalloc, or any other routines which can throw an
 *   exception when called, you can set this option for improved
 *   performance.
 *
 * SS_OPTION_RUNTIME_EXCEPTION_FREE_CODE - Similar to
 *   SS_OPTION_EXCEPTION_FREE_CODE except it only applies to the "run-time"
 *   routines: mdlGetTimeOfNextVarHit, mdlOutputs, mdlUpdate, and
 *   mdlDerivatives.
 *
 * SS_OPTION_DISCRETE_VALUED_OUTPUT - This should be specified if your
 *   S-function has a discrete valued outputs.  This is checked when
 *   your S-function is placed within an algebraic loop. If your S-function
 *   has discrete valued outputs, then its outputs will not be assigned
 *   algebraic variables.
 *
 * SS_OPTION_PLACE_ASAP - This is used to specify that your S-function
 *   should be placed as soon as possible. This is typically used by
 *   devices connecting to hardware.
 *
 * SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION - This is used to specify
 *   that the input to your S-function input ports can be either 1 or
```

```
     *   the size specified by the port which is usually referred to as
     *   the block width.
     *
     * SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME - This is used to disable
     *   your S-function block from inheriting a constant sample time.
     *
     * SS_OPTION_ASYNCHRONOUS - This option applies only to S-functions that
     *   have no input ports and 1 output port.  The output port must be
     *   configured to perform function calls on every element.  If any of
     *   these requirements are not met, the SS_OPTION_ASYNCHRONOUS is
     *   ignored.  Use this option when driving function-call subsystems that
     *   will be attached to interupt service routines.
     *
     * SS_OPTION_ASYNC_RATE_TRANSITION - Use this when your s-function converts
     *   a signal from one rate to another rate.
     *
     * SS_OPTION_RATE_TRANSITION - Use this why your S-function is behaving
     *   as a unit delay or ZOH. This option is only supported for these two
     *   operations. An unit delay operation is identified by the presence
     *   of mdlUpdate and if not present then the operation is ZOH.
     */

    ssSetOptions(          S, 0);   /* general options (SS_OPTION_xx)        */

} /* end mdlInitializeSizes */
```

For an example, see the `timestwo` S-function on page 3-7.

Masked Multiport S-Functions

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

```
set_param('block','MaskSelfModifiable','on')
```

at the MATLAB prompt. Failure to specify that the mask modifies the appearance of the block will mean that when the library link is loaded, the number of ports will not always match what is shown in the library.

Configuring Input and Output Port Properties

In `mdlInitializeSizes`, you can specify the port widths as a positive nonzero integer. In this case, the corresponding input/output signal will have the specified width when accessed by your S-function routines during the simulation loop. However, if you specify `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION` and a positive nonzero integer, `N`, for an input port width, then the width of the corresponding signal will be

either 1 or N. You can use ssGetInputPortWidth in the routines called during the simulation loop to determine the actual port width being used.

You can also specify that input and/or output signals are dynamically sized (via the DYNAMICALLY_SIZED define). Simulink determines the corresponding widths when it propagates vector widths throughout your model. You can influence the vector width propagation via the mdlSetInputPortWidth and mdlSetOutputPort width routines. If your S-function doesn't specify these routines, then the *default scalar expansion rules* apply.

The best way to understand the default scalar expansion rules is to consider a an S-function implementing a sum block with three input ports. The block has an overall width, called the *block width*, equal to the width of the output port. Each input port can have a width of 1 or the block width providing that at least one input port has a width equal to the block width. Each dynamically sized work vector will be set to the block width. See *matlabrooot*/simulink/src/sfun_multiport.c for an example.

If this handling of dynamically sized input and output signals doesn't meet your needs, then you need to use the optional routines mdlSetInputPortWidth and mdlSetOutputPortWidth.

Note You must correctly specify the number of ports before setting any properties. If you attempt to set a property of a port that doesn't exist, you will be accessing invalid memory and Simulink will crash.

---

mdlSetInputPortSampleTime

This routine is called with the candidate sample time for an inherited sample time input port. If the proposed sample time is acceptable, the routine should go ahead and set the actual port sample time using ssSetInputPortSampleTime. If the sample time is unacceptable an error should generated via ssSetErrorStatus. Note that any other inherited input or output ports whose sample times are implicitly defined by the sample time of the given port can also have their widths set via calls to ssSetInputPortSampleTime or ssSetOutputPortSampleTime.

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

- continuous: `[0.0, 0.0]`
- discrete: `[period, offset]` where `0.0 < period < inf` and `0.0 <= offset < period`

Constant, triggered, and variable step sample times will not be propagated to S-functions with port based sample times.

This is the code synopsis for `mdlSetInputPortSampleTime`:

```
#if defined(MDL_SET_INPUT_PORT_SAMPLE_TIME) && defined(MATLAB_MEX_FILE)

static void mdlSetInputPortSampleTime(SimStruct *S,
                                      int_T     portIdx,
                                      real_T    sampleTime,
                                      real_T    offsetTime)
  {
  }
#endif /* MDL_SET_INPUT_PORT_SAMPLE_TIME */
```

See *`matlabrooot`*`/simulink/src/sfun_multirate.c` for an example.

mdlSetOutputPortSampleTime

This routine is called with the candidate sample time for an inherited sample time output port. If the proposed sample time is acceptable, the routine sets the actual port sample time using `ssSetOutputPortSampleTime`. If the sample time is unacceptable an error is generated by `ssSetErrorStatus`. Note that any other inherited input or output ports whose sample times are implicitly defined by the sample time of the given port can also have their widths set by calls to `ssSetInputPortSampleTime` or `ssSetOutputPortSampleTime`.

Normally, sample times are propagated forwards. However, if sources feeding this block have an inherited sample time, then Simulink may choose to back-propagate known sample times to this block. When back-propagating sample times, Simulink calls this routine in succession for all inherited output port signals.

This is the code synopsis for `mdSetOutputPortSampleTime`:

```
#if defined(MDL_SET_OUTPUT_PORT_SAMPLE_TIME) && defined(MATLAB_MEX_FILE)
static void mdlSetOutputPortSampleTime(SimStruct *S,
                                       int_T     portIdx,
                                       real_T    sampleTime,
                                       real_T    offsetTime)
  {
  }
#endif /* MDL_SET_OUTPUT_PORT_SAMPLE_TIME */
```

See *matlabrooot*`/simulink/src/sfun_multirate.c` for an example.

mdlSetInputPortWidth

During vector-width propagation Simulink calls `mdlSetInputPortWidth` with the candidate width for a dynamically sized port. If the proposed width is acceptable, the routine sets the actual port width using `ssSetInputPortWidth`. If the size is unacceptable an error is generated by `ssSetErrorStatus`. Note that any other dynamically sized input or output ports whose widths are implicitly defined by the width of the given port can also have their widths set by calls to `ssSetInputPortWidth` or `ssSetOutputPortWidth`.

The synopsis is:

```
#define MDL_SET_INPUT_PORT_WIDTH   /* Change to #undef to remove function. */
#if defined(MDL_SET_INPUT_PORT_WIDTH) && defined(MATLAB_MEX_FILE)
void mdlSetInputPortWidth(SimStruct *S, int portIndex, int width)
{
}
#endif /* MDL_SET_INPUT_PORT_WIDTH */
```

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

mdlSetOutputPortWidth

`mdlSetOutputPortWidth` works in the same way with output ports as `mdlSetInputPortWidth` does with input ports.

The synopsis is:

```
#define MDL_SET_OUTPUT_PORT_WIDTH   /* Change to #undef to remove function */
#if defined(MDL_SET_OUTPUT_PORT_WIDTH) && defined(MATLAB_MEX_FILE)
void mdlSetOutputPortWidth(SimStruct *S, int portIndex, int width)
{
}
#endif /* MDL_SET_OUTPUT_PORT_WIDTH */
```

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

Setting Sample Times for C MEX S-Functions

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

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

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

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

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

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

Block-based Sample Times

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

- `mdlInitializeSizes`
- `mdlInitializeSampleTimes`

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

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

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

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

Setting Sample Times and Specifying Function Calls in mdlInitializeSampleTimes. `mdlInitializeSampleTimes` is used to specify two pieces of execution information:

- Sample and offset times — In `mdlInitializeSizes`, specify the number of sample times you'd like your S-function to have by using the `ssSetNumSampleTimes` macro. In `mdlInitializeSampleTimes`, you must specify the sampling period and offset for each sample time.

  Sample times can be a function of the input/output port widths. In `mdlInitializeSampleTimes`, you can specify that sample times are a function of `ssGetInputPortWidth` and `ssGetGetOutputPortWidth`.
- Function calls — In `ssSetCallSystemOutput`, specify which output elements are performing function calls. See *`matlabroot`*`/simulink/src/sfun_fcncall.c` for an example.

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

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

where *`sampleTimePairIndex`* starts at 0.

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

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

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

```
[INHERITED_SAMPLE_TIME,  0.0                          ]
```

or

```
[INHERITED_SAMPLE_TIME,  FIXED_IN_MINOR_STEP_OFFSET]
```

The following guidelines may help aid in specifying sample times:

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

  `[`*`discrete_sample_period, offset`*`]`

  where

  `discrete_sample_period` `> 0.0`

  and

  `0.0 <=` *`offset`* `<` *`discrete_sample_period`*

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

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

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

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

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

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

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

The synopsis is:

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
} /* End of mdlInitializeSampleTimes. */
```

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

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

Port-based Sample Times

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

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

Specifying the Number of Sample Times in mdlInitializeSizes. To specify port-based sample times, use

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

with:

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

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

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

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

---

mdlSetInputPortSampleTime. Simulink calls this routine with the candidate sample time for an inherited sample time input port. If the proposed sample time is acceptable, the routine sets the actual port sample time using `ssSetInputPortSampleTime`. If the sample time is unacceptable `ssSetErrorStatus` generates an error. If any other inherited input or output ports have sample times that are implicitly defined by the sample time of the given port, then you can specify their widths by calls to `ssSetInputPortSampleTime` or `ssSetOutputPortSampleTime`.

When you specify port based sample times as inherited, the sample time will be one of the following:

- continuous: `[0.0, 0.0]`
- discrete: [*period*, *offset*] where `0.0 <` *period* `< inf` and `0.0 <=` *offset* `<` *period*

Constant, triggered, and variable step sample times will not be propagated to S-functions with port based sample times.

This is the code synopsis for `mdlSetInputPortSampleTime`:

```
#if defined(MDL_SET_INPUT_PORT_SAMPLE_TIME) && defined(MATLAB_MEX_FILE)

  static void mdlSetInputPortSampleTime(SimStruct *S,
                                        int_T     portIdx,
                                        real_T    sampleTime,
                                        real_T    offsetTime)
  {
  }
#endif /* MDL_SET_INPUT_PORT_SAMPLE_TIME */
```

See *matlabrooot*`/simulink/src/sfun_multirate.c` for an example.

mdlSetOutputPortSampleTime. This routine is called with the candidate sample time for an inherited sample time output port. If the proposed sample time is acceptable, the routine sets the actual port sample time using `ssSetOutputPortSampleTime`. If the sample time is unacceptable an error is generated by `ssSetErrorStatus`. Any other inherited input or output ports whose sample times are implicitly defined by the sample time of the given port can also have their widths set by calls to `ssSetInputPortSampleTime` or `ssSetOutputPortSampleTime`.

Normally, sample times are propagated forwards. However, if sources feeding this block have an inherited sample time, then Simulink may choose to back-propagate known sample times to this block. When back-propagating sample times, Simulink calls this routine in succession for all inherited output port signals.

This is the code synopsis for `mdSetOutputPortSampleTime`:

```
#if defined(MDL_SET_OUTPUT_PORT_SAMPLE_TIME) && defined(MATLAB_MEX_FILE)
  static void mdlSetOutputPortSampleTime(SimStruct *S,
                                         int_T     portIdx,
                                         real_T    sampleTime,
                                         real_T    offsetTime)
  {
  }
#endif /* MDL_SET_OUTPUT_PORT_SAMPLE_TIME */
```

See *matlabrooot*`/simulink/src/sfun_multirate.c` for an example.

Multirate S-Function Blocks

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

```
ssIsSampleHit(S, st_index, tid)
```

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

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

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

In the `mdlUpdate` function, the following statement would encapsulate the code that defines the behavior for the sample time of 0.75 seconds:

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

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

Example - Defining a Sample Time for a Continuous Block. This example defines a sample time for a block that is continuous in nature.

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

You must add this statement to the `mdlInitializeSizes` function:

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

Example - Defining a Sample Time for a Hybrid Block. This example defines sample times for a hybrid S-Function block.

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

  /* Discrete state sample time and offset. */
  ssSetSampleTime(S, 1, 0.1);
  ssSetOffsetTime(S, 1, 0.025);
}
```

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

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

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

Configuring Work Vectors

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

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

Work vectors have several advantages:

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

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

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

Table 3-4: Macros Used in Specifying Vector Widths

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

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

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

If the default behavior of dynamically sized vectors does not meet your needs, use `mdlSetWorkWidths` and the macros listed in Table 3-4: "Macros Used in Specifying Vector Widths" on page 3–44 to set explicitly the sizes of the work vectors. Also, `mdlSetWorkWidths` allows you to set your work vector lengths as a function of the block sample time and/or port widths.

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

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

Work Vectors and Zero Crossings

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

the mdlZeroCrossings routine to return the nonsampled zero crossings. See *matlabroot*/simulink/src/sfun_zc.c for an example.

mdlSetWorkWidths

The optional simulation only routine, mdlSetWorkWidths, is called after input port width, output port width, and sample times of the S-function have been determined to set any state- and work-vector sizes that are a function of the input, output, and/or sample times. This routine is used to specify the nonzero work vector widths via the macros ssNumContStates, ssSetNumDiscStates, ssSetNumRWork, ssSetNumIWork, ssSetNumPWork, ssSetNumModes, and ssSetNumNonsampledZCs.

An Example Involving a Pointer Work Vector

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

The statement below, included in the mdlInititalizeSizes function, indicates that the pointer-work vector is to contain one element:

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

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

```
#define MDL_START  /* Change to #undef to remove function. */
#if defined(MDL_START)
static void mdlStart(real_T *x0, SimStruct *S)
{
  FILE *fPtr;
  void **PWork = ssGetPWork(S);
  fPtr = fopen("file.data", "r");
  PWork[0] = fPtr;
}
#endif /*  MDL_START */
```

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

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

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

The synopsis is:

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

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

Memory Allocation

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

routines. In `mdlStart` allocate and initialize the memory and place the pointer to it either in pointer-work vector elements

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

or attach it as user data:

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

In `mdlTerminate`, free the allocated memory.

S-Function Initialization

The S-function routines discussed so far have dealt with defining the S-function characteristics, such as the number of input and output ports and their widths. This section describes how to initialize S-functions. This may mean initializing the work vectors or the output signal of the S-function. With the Real-Time Workshop, this may mean initializing hardware. The routines, `mdlStart` and `mdlInitializeConditions`, perform initialization.

mdlStart

The optional routine, `mdlStart`, is called once at the start of model execution. Place one time initialization code here.

The synopsis is:

```
#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
static void mdlStart(SimStruct *S)
{
}
#endif /*  MDL_START */
```

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

mdlInitializeConditions

You can initialize the continuous and discrete states for your S-function block in the optional S-function routine `mdlInitializeConditions`. The initial states are placed in the state vector, `ssGetContStates(S)` and/or `ssGetNumDiscStates(S)`. You can also perform any other initialization activities that your S-function may require.

Simulink will call this routine in two places:

- At the start of simulation
- If it is present in an enabled subsystem configured to reset states. In this case, Simulink will call `mdlInitializeConditions` when the enabled subsystem restarts execution to reset the states.

You can use the `ssIsFirstInitCond(S)` macro to determine whether the current call is the first to `mdlInitializeConditions`.

The synopsis is:

```
#define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
#if defined(MDL_INITIALIZE_CONDITIONS)
static void mdlInitializeConditions(SimStruct *S)
{
}
#endif /* MDL_INITIALIZE_CONDITIONS */
```

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

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

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

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

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

}
#endif /* MDL_INITIALIZE_CONDITIONS */
```

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

S-Function Routines Called During Simulation

Conceptually, the way Simulink performs a simulation is that it executes the blocks in your model in the order dictated by their connections. After executing this list of blocks, the process is repeated until the end of simulation is reached. This repeated execution of the blocks is referred to as the *simulation loop*.

Your S-functions interact with the Simulink during simulations by the S-function routines outlined in the simulation loop of the flowchart presented on page 3-16.

mdlGetTimeOfNextVarHit

The optional routine, `mdlGetTimeOfNextVarHit`, to be used with variable step solvers, is called to get the time of the next variable sample time hit. This function is called once for every major integration time step. It must return the time of the next hit by using `ssSetTNext`. The time of the next hit must be greater than `ssGetT(S)`.

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

The synopsis is:

```
#define MDL_GET_TIME_OF_NEXT_VAR_HIT  /* Change to #undef to remove function */
#if defined(MDL_GET_TIME_OF_NEXT_VAR_HIT) && \
   (defined(MATLAB_MEX_FILE) || defined(NRT))
static void mdlGetTimeOfNextVarHit(SimStruct *S)
{
  ssSetTNext(S, <timeOfNextHit>);
}
#endif /* MDL_GET_TIME_OF_NEXT_VAR_HIT */
```

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

mdlOutputs

In the `mdlOutputs` routine, you compute the outputs of your S-function block. Generally outputs are placed in the output vector(s), `ssGetOutputPortSignal`.

The synopsis is:

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
} /* end mdlOutputs */
```

The `tid` (task ID) argument is used in conjunction with multirate S-functions to determine when a particular task has a sample hit.

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

mdlUpdate

The optional routine `mdlUpdate` 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.

The `mdlUpdate` routine is called if you include this code in your S-function's main module:

```
#define MDL_UPDATE  /* Change to #undef to remove function. */
#if defined(MDL_UPDATE)
static void mdlUpdate(SimStruct *S, int_T tid)
{
}
#endif /* MDL_UPDATE */
```

Use this code if your S-function has one or more discrete states or does *not* have direct feedthrough.

The reason for this is that most S-functions that do not have discrete states but do have direct feedthrough do not have update functions. Therefore, Simulink is able to eliminate the need for the extra call in these circumstances.

If your S-function needs to have its `mdlUpdate` routine called and it does not satisfy either of the above two conditions, specify that it has a discrete state using the `ssSetNumDiscStates` macro in the `mdlInitializeSizes` function.

The `tid` (task ID) argument is used in conjunction with multirate S-functions to determine when a particular task has a sample hit.

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

mdlDerivatives

In the optional routine, `mdlDerivatives`, you compute the S-Function block's derivatives for its continuous states. The derivatives are placed in the derivative vector, `ssGetdX(S)`. `mdlDerivatives` is called only during minor time steps.

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

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

The synopsis is:

```
#define MDL_DERIVATIVES /* Change to #undef to remove function. */
#if defined(MDL_DERIVATIVES)
static void mdlDerivatives(SimStruct *S)
{
}
#endif /* MDL_DERIVATIVES */
```

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

mdlZeroCrossings

You can use the optional `mdlZeroCrossings` routine, when your S-function has registered the `CONTINUOUS_SAMPLE_TIME` and has nonsampled zero crossings (`ssGetNumNonsampledZCs(S) > 0`). The `mdlZeroCrossings` routine is used to provide Simulink with signals that are to be tracked for zero crossings. These are typically:

- Continuous signals entering the S-function
- Internally generated signals that cross zero when a discontinuity would normally occur in `mdlOutputs`

Thus, the zero crossing signals are used to locate the discontinuities and end the current time step at the point of the zero crossing. To provide Simulink with zero crossing signal(s), `mdlZeroCrossings` updates the `ssGetNonsampleZCs(S)` vector.

The synopsis is:

```
#define MDL_ZERO_CROSSINGS   /* Change to #undef to remove function. */
#if defined(MDL_ZERO_CROSSINGS) && (defined(MATLAB_MEX_FILE) || defined(NRT))
static void mdlZeroCrossings(SimStruct *S)
{
}
#endif /* MDL_ZERO_CROSSINGS */
```

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

mdlTerminate

In the `mdlTerminate` routine, perform any actions that are necessary at the termination of a simulation. For example, if memory was allocated in `mdlInitializeSizes` or `mdlStart`, this is the place to free it. Suppose your S-function allocates a few chunks of memory and saves them in `PWork`. The following code fragment would free this memory:

```
static void mdlTerminate(SimStruct *S)
{
  int i;
  for (i = 0; i<ssGetNumPWork(S); i++) {
    if (ssGetPWorkValue(S,i) != NULL) {
      free(ssGetPWorkValue(S,i));
    }
  }
}
```

Using S-Functions With the Real-Time Workshop

In general, you can use S-functions in the Real-Time Workshop. However, certain cases require these modifications to S-functions:

- Setting names of extra modules used in building your S-function
- Setting `RTWdata` for the S-function
- Adding an `mdlRTW` function

S-Function Module Names for RTW Build's

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

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

then specify the names of the modules without the extension using the command:

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

The parameter can also be a variable as in

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

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

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

S-Function RTWdata for Generating Code with RTW

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

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

produces this result:

```
ans =

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

Inside the *`model`*`.rtw` for the associated S-function block is this information:

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

mdlRTW

The `mdlRTW` routine can help you inline (embed) your S-function in the generated code. Inlining means to embed in your code in Real-Time Workshop generated code. Typically, this is done for performance reasons. You may also be required to inline your S-function if you have an `mdlProcessParameters`

routine or your S-function has a “simulation mode” and a “real-time mode” such as a hardware I/O S-function that simulates the I/O device in Simulink and interacts with the I/O device in real-time.

To inline an S-function in the generated code, you must use the Target Language Compiler. Details on this can be found in the *Real-Time Workshop User's Guide* and *Target Language Compiler Reference Guide*. Inlining of S-functions alters the code generation process by adding information to the generated *model*`.rtw` file. This is achieved by adding an `mdlRTW` routine to your S-function. See `simulink/src/sfuntmpl.doc` for more details. For an example, see `simulink/src/sfun_multiport.c` and `toolbox/simulink/blocks/sfun_multiport.tlc`.

Examples of C MEX-File S-Function Blocks

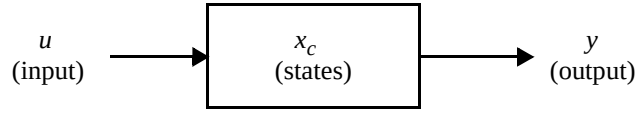
Most S-Function blocks require the handling of states, continuous or discrete. The following sections discuss common types of systems that you can model in Simulink with S-functions:

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

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

Example - Continuous State S-Function

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



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

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

S-functions that contain continuous states implement a state-space equation. The output portion is placed in `mdlOutputs` and the derivative portion in `mdlDerivatives`. To visualize how the integration works, refer back to the flowchart on page 3-16. The output equation above corresponds to the `mdlOutputs` in the major time step. Next, the example enters the integration section of the flowchart. Here Simulink performs a number of minor time steps

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

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

matlabroot/simulink/src/csfunc.c

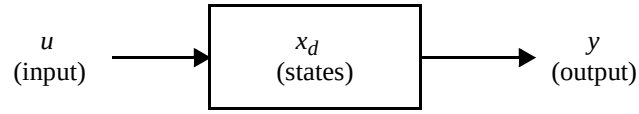
```
/*  File    : csfunc.c
 *  Abstract:
 *
 *      Example C-MEX S-function for defining a continuous system.
 *
 *      x' = Ax + Bu
 *      y  = Cx + Du
 *
 *      For more details about S-functions, see simulink/src/sfuntmpl.doc.
 *
 *  Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 *  $Revision: 1.2 $
 */

#define S_FUNCTION_NAME csfunc
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

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

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

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

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

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


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

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

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

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

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

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

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

/* Function: mdlOutputs =======================================================
 * Abstract:
 *      y = Cx + Du
 */
```

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    real_T            *x    = ssGetContStates(S);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

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

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

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

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

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

Example - Discrete State S-Function

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



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

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

`dsfunc.c` implements a discrete state-space equation. The output portion is placed in `mdlOutputs` and the update portion in `mdlUpdate`. To visualize how the simulation works, refer to the flowchart on page 3-16. The output equation above corresponds to the `mdlOutputs` in the major time step. The update equation above corresponds to the `mdlUpdate` in the major time step. If your model does not contain continuous elements, the integration phase is skipped and time is moved forward to the next discrete sample hit.

matlabroot/simulink/src/dsfunc.c

```
/*  File    : dsfunc.c
 *  Abstract:
 *
 *      Example C MEX 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.
 *
 *  Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 *  $Revision: 1.3 $
 */

#define S_FUNCTION_NAME dsfunc
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

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

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

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

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

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


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

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

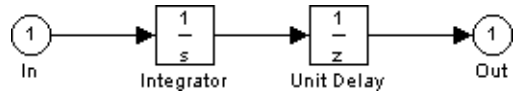
```
        return; /* Parameter mismatch will be reported by Simulink */
    }

    ssSetNumContStates(S, 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 */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}



/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    Specify that we inherit our sample time from the driving block.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, 1.0);
    ssSetOffsetTime(S, 0, 0.0);
}



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

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

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

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



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

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

    x[0]=tempX[0];
    x[1]=tempX[1];
}
/* Function: mdlTerminate =====================================================
 * Abstract:
 *    No termination needed, but we are required to have this routine.
 */
static void mdlTerminate(SimStruct *S)
{
}

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

Example - Hybrid System S-Functions

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

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



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

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

matlabroot/simulink/src/mixedm.c

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

#include "simstruc.h"

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

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

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

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

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

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

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

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

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

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

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

    xC0[0] = 1.0;
    xD0[0] = 1.0;
}

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

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

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

```
    real_T *xD = ssGetRealDiscStates(S);
    real_T *xC = ssGetContStates(S);

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

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

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

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

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

Example - Variable Step S-Function

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

This example demonstrates how to correctly work with the fixed and variable step solvers when the equations (functions) that are being integrated change

during the simulation. In the transfer function used in this example, the parameters of the transfer function vary with time.

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

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

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

matlabroot/simulink/src/vsfunc.c

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

#define S_FUNCTION_NAME vsfunc
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

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



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

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

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

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

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

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



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



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

    x0[0] = 0.0;
}



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

    /* Make sure input will increase time */
    if (U(1) <= 0.0) {
        /* If not, abort simulation */
        ssSetErrorStatus(S,"Variable step control input must be "
                           "greater than zero");
        return;
    }
  ssSetTNext(S, ssGetT(S)+U(1));
}
```

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

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



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

    x[0]=U(0);
}



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

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

Example - Zero Crossing S-Function

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

matlabroot/simulink/src/sfun_zc_sat.c

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


#define S_FUNCTION_NAME  sfun_zc_sat
#define S_FUNCTION_LEVEL 2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#if defined(MATLAB_MEX_FILE)
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink */
    }
#endif

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

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

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

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

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

/*
 * inputs
 *   If the upper or lower limits are not scalar then
 *   the input is set to the same size.  However, the
 *   ssSetOptions below allows the actual width to
 *   be reduced to 1 if needed for scalar expansion.
 */
if (!ssSetNumInputPorts(S, 1)) return;

ssSetInputPortDirectFeedThrough(S, 0, 1 );

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

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

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

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

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

} /* end mdlInitializeSizes */



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

```
#define     MDL_SET_WORK_WIDTHS
#if defined(MDL_SET_WORK_WIDTHS) && defined(MATLAB_MEX_FILE)
/* Function: mdlSetWorkWidths =================================================
 *   The width of the modes and the zero crossings 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 vise-versa.
         */
        nModes         = numOutput;
        nNonsampledZCs = 2 * numOutput;
    } else {
        nModes         = 0;
        nNonsampledZCs = 0;
    }
    ssSetNumModes(S,nModes);
    ssSetNumNonsampledZCs(S,nNonsampledZCs);
}
#endif /* MDL_SET_WORK_WIDTHS */

/* Function: mdlOutputs =======================================================
 * Abstract:
 *
 *  A saturation is described by three equations.
 *
 *    (1)     y = UpperLimit
 *    (2)     y = u
 *    (3)     y = LowerLimit
 *
 *  When this block is used with a fixed-step solver or it has a noncontinuous
 *  sample time, the equations are used as is.
 *
 *  Now consider the case of this block being used with a variable step solver
```

```
 *  and having a continuous sample time. Solvers work best on smooth problems.
 *  In order for the solver to work without chattering, limit cycles, or
 *  similar problems, it is absolutely crucial that the same equation be used
 *  throughout the duration of a MajorTimeStep. To visualize this, consider
 *  the case of the Saturation block feeding an Integrator block.
 *
 *  To implement this rule, the mode vector is used to specify the
 *  valid equation based on the following:
 *
 *    if                          UpperLimit < u    then   use (1)
 *    if       LowerLimit <= u <= UpperLimit        then   use (2)
 *    if   u < LowerLimit                           then   use (3)
 *
 *  The mode vector is changed only at the beginning of a MajorTimeStep.
 *
 *  During a minor time step, the equation specified by the mode vector
 *  is used without question.  Most of the time, the value of u will agree
 *  with the equation specified by the mode vector.  However, sometimes u's
 *  value will indicate a different equation.  Nonetheless, the equation
 *  specified by the mode vector must be used.
 *
 *  When the mode and u indicate different equations, the corresponding
 *  calculations are not correct.  However, this is not a problem.  From
 *  the ZC function, the solver will know that an equation switch occurred
 *  in the middle of the last MajorTimeStep.  The calculations for that
 *  time step will be discarded.  The ZC function will help the solver
 *  find the exact instant at which the switch occurred.  Using this knowledge,
 *  the length of the MajorTimeStep will be reduced so that only one equation
 *  is valid throughout the entire time step.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs     = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y        = ssGetOutputPortRealSignal(S,0);
    int_T             numOutput = ssGetOutputPortWidth(S,0);
    int_T             iOutput;

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

    if (ssGetNumNonsampledZCs(S) == 0) {
        /*
         * This block is being used with a fixed-step solver or it has
         * a noncontinuous sample time, so we always saturate.
         */
```

```
        for (iOutput = 0; iOutput < numOutput; iOutput++) {
            if (*uPtrs[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, that is, the valid equation for each output scalar.
             */
            for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {
                if ( *uPtrs[uIdx] > *upperLimit ) {
                    /*
                     * Upper limit eq is valid.
                     */
                    mode[iOutput] = UpperLimitEquation;
                } else if ( *uPtrs[uIdx] < *lowerLimit ) {
                    /*
                     * Lower limit eq is valid.
                     */
                    mode[iOutput] = LowerLimitEquation;
                } else {
                    /*
                     * Nonlimit eq is valid.
                     */
                    mode[iOutput] = NonLimitEquation;
                }
                /*
                 * Adjust indices to give scalar expansion if needed.
                 */
                uIdx       += uInc;
```

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

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

        } /* end IsMajorTimeStep */

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

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

} /* end mdlOutputs */
```

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

/* Function: mdlZeroCrossings =====================================================
 * Abstract:
 *  This will only be called if the number of nonsampled zero crossings is
 *  greater than 0, which means this block has a continuous sample time and the
 *  the model is using a variable step solver.
 *
 *  Calculate ZC signals that help the solver find the
 *  exact instants at which equation switches occur:
 *
 *    if                          UpperLimit < u    then   use (1)
 *    if       LowerLimit <= u <= UpperLimit        then   use (2)
 *    if   u < LowerLimit                           then   use (3)
 *
 *  The key words are help find. There is no choice of a function that will
 *  direct the solver to the exact instant of the change. The solver will
 *  track the zero crossing signal and do a bisection style search for the
 *  exact instant of equation switch.
 *
 *  There is generally one ZC signal for each pair of signals that can
 *  switch.  The three equations above would break into two pairs (1)&(2)
 *  and (2)&(3).  The  possibility of a "long jump" from (1) to (3) does
 *  not need to be handled as a separate case.  It is implicitly handled.
 *
 *  When a ZCs are calculated, the value is normally used twice.  When it is
 *  first calculated, it is used as the end of the current time step.  Later,
 *  it will be used as the beginning of the following step.
 *
 *  The sign of the ZC signal always indicates an equation from the pair.  In the
 *  context of S-functions, which equation is associated with a positive ZC and
 *  which is associated with a negative ZC doesn't really matter.  If the ZC is
 *  positive at the beginning and at the end of the time step, this implies that the
 *  positive equation was valid throughout the time step.  Likewise, if the
 *  ZC is negative at the beginning and at the end of the time step, this
 *  implies that the negative equation was valid throughout the time step.
 *  Like any other nonlinear solver, this is not fool proof, but it is an
 *  excellent indicator.  If the ZC has a different sign at the beginning and
 *  at the end of the time step, then a equation switch definitely occured
 *  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 occured
 *  25% = 100%*(-2)/(-2-(+6)) of the way into the time step.  It will almost
 *  never be true that 25% is perfectly correct.  There is no perfect choice
 *  for a ZC signal, but there are some good rules.  First, choose the ZC
 *  signal to be continuous.  Second, choose the ZC signal to give a monotonic
 *  measure of the "distance" to a signal switch; strictly monotonic is ideal.
 */
```

```
static void mdlZeroCrossings(SimStruct *S)
{
    int_T             iOutput;
    int_T             numOutput = ssGetOutputPortWidth(S,0);
    real_T            *zcSignals = ssGetNonsampledZCs(S);
    InputRealPtrsType uPtrs      = ssGetInputPortRealSignalPtrs(S,0);

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

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

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

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

        /*
         * Adjust indices to give scalar expansion if needed.
         */
        uIdx       += uInc;
        upperLimit += upperLimitInc;
        lowerLimit += lowerLimitInc;
```

```
    }
}

#endif /* end mdlZeroCrossings */

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

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

Example - Time Varying Continuous Transfer Function

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

matlabroot/simulink/src/stvctf.c

```
/*
 * File : stvctf.c
 * Abstract:
 *      Time Varying Continuous Transfer Function block
 *
 *      This S-function implements a continuous time transfer function
 *      whose transfer function polynomials are passed in via the input
 *      vector.  This is useful for continuous time adaptive control
 *      applications.
 *
 *      This S-function is also an example of how to "use banks" to avoid
 *      problems with computing derivatives when a continuous output has
 *      discontinuities. The consistency checker can be used to verify that
 *      your S-function is correct with respect to always maintaining smooth
 *      and consistent signals for the integrators. By consistent we mean that
 *      two mdlOutput calls at major time t and minor time t are always the
 *      same. The consistency checker is enabled on the diagnostics page of the
 *      simulation parameters dialog box. The update method of this S-function
 *      modifies the coefficients of the transfer function, which cause the
 *      output to "jump." To have the simulation work properly, we need to let
 *      the solver know of these discontinuities by setting
 *      ssSetSolverNeedsReset. 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 used to verify that when
 *      the solver needs to try a smaller step size that 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, are delayed until after
 *      the integrate phase is complete.
 *
 * See simulink/src/sfuntmpl.doc.
 *
 * Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 * $Revision: 1.8 $
```

```
 */

#define S_FUNCTION_NAME  stvctf
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

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


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

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

      if (mxGetNumberOfElements(NUM(S)) > mxGetNumberOfElements(DEN(S)) &&
          mxGetNumberOfElements(DEN(S)) > 0  && *mxGetPr(DEN(S)) != 0.0) {
          ssSetErrorStatus(S,"The denominator must be of higher order than "
```

```
                            "the numerator, nonempty and with first "
                            "element nonzero");
            return;
        }

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


/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    int_T nContStates;
    int_T nCoeffs;

    /* See sfuntmpl.doc for more details on the macros below. */

    ssSetNumSFcnParams(S, NPARAMS);  /* Number of expected parameters. */
#if defined(MATLAB_MEX_FILE)
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink. */
    }
#endif


    /*
     * Define the characteristics of the block:
     *
     *   Number of continuous states:     length of denominator - 1
     *   Inputs port width                2 * (NumContStates+1) + 1
     *   Output port width                1
     *   DirectFeedThrough:               0 (Although this should be computed.
     *                                       We'll assume coefficients entered
     *                                       are strictly proper).
     *   Number of sample times:          2 (continuous and discrete)
     *   Number of Real work elements:    4*NumCoeffs
     *                                    (Two banks for num and den coeff's:
     *                                     NumBank0Coeffs
     *                                     DenBank0Coeffs
```

```
     *                                     NumBank1Coeffs
     *                                     DenBank1Coeffs)
     *   Number of Integer work elements: 2 (indicator of active bank 0 or 1
     *                                       and flag to indicate when banks
     *                                       have been updated).
     *
     * The number of inputs arises from the following:
     *   o 1 input (u)
     *   o the numerator and denominator polynomials each have NumContStates+1
     *     coefficients
     */
    nCoeffs     = mxGetNumberOfElements(DEN(S));
    nContStates = nCoeffs - 1;

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

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

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

    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 */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);

} /* end mdlInitializeSizes */

/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *      This function is used to specify the sample time(s) for the
 *      S-function.  This S-function has two sample times.  The
 *      first, a continuous sample time, is used for the input to the
 *      transfer function, u.  The second, a discrete sample time
 *      provided by the user, defines the rate at which the transfer
 *      function coefficients are updated.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    /*
     * the first sample time, continuous
     */
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
```

```
    ssSetOffsetTime(S, 0, 0.0);

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

} /* end mdlInitializeSampleTimes */

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

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

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

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

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

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

```
        }
    }

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

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

} /* end mdlInitializeConditions */



/* Function: mdlOutputs =======================================================
 * Abstract:
 *      The outputs for this block are computed by using a controllable state-
 *      space representation of the transfer function.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    if (ssIsContinuousTask(S,tid)) {
        int               i;
        real_T            *num;
        int               nContStates = ssGetNumContStates(S);
        real_T            *x          = ssGetContStates(S);
        int_T             nCoeffs     = nContStates + 1;
        InputRealPtrsType uPtrs       = ssGetInputPortRealSignalPtrs(S,0);
        real_T            *y          = ssGetOutputPortRealSignal(S,0);
        int_T             *activeBank = ssGetIWork(S);

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

```
        /*
         * The continuous system is evaluated using a controllable state space
         * representation of the transfer function.  This implies that the
         * output of the system is equal to:
         *
         *     y(t) = Cx(t) + Du(t)
         *          = [ b1 b2 ... bn]x(t) + b0u(t)
         *
         * where b0, b1, b2, ... are the coefficients of the numerator
         * polynomial:
         *
         *    B(s) = b0 s^n + b1 s^n-1 + b2 s^n-2 + ... + bn-1 s + bn
         */
        *y = *num++ * (*uPtrs[0]);
        for (i = 0; i < nContStates; i++) {
            *y += *num++ * *x++;
        }
    }

} /* end mdlOutputs */


#define MDL_UPDATE
/* Function: mdlUpdate ======================================================
 * Abstract:
 *      Every time through the simulation loop, update the
 *      transfer function coefficients. Here we update the oldest bank.
 */
static void mdlUpdate(SimStruct *S, int_T tid)
{
    if (ssIsSampleHit(S, 1, tid)) {
        int_T             i;
        InputRealPtrsType uPtrs        = ssGetInputPortRealSignalPtrs(S,0);
        int_T             uIdx         = 1;/*1st coeff is after signal input*/
        int_T             nContStates  = ssGetNumContStates(S);
        int_T             nCoeffs      = nContStates + 1;
        int_T             bankToUpdate = !ssGetIWork(S)[0];
        real_T            *num         = ssGetRWork(S)+bankToUpdate*2*nCoeffs;
        real_T            *den         = num + nCoeffs;

        real_T            den0;
        int_T             allZero;


        /*
         * Get the first denominator coefficient.  It will be used
         * for normalizing the numerator and denominator coefficients.
         *
         * If all inputs are zero, we probably could have unconnected
         * inputs, so use the parameter as the first denominator coefficient.
         */
        den0 = *uPtrs[uIdx+nCoeffs];
```

```
if (den0 == 0.0) {
    den0 = mxGetPr(DEN(S))[0];
}

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

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

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

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

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

/*
```

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

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

} /* end mdlUpdate */



#define MDL_DERIVATIVES
/* Function: mdlDerivatives =============================================
 * Abstract:
 *      The derivatives for this block are computed by using a controllable
 *      state-space representation of the transfer function.
 */
static void mdlDerivatives(SimStruct *S)
{
    int_T             i;
    int_T             nContStates = ssGetNumContStates(S);
    real_T            *x          = ssGetContStates(S);
    real_T            *dx         = ssGetdX(S);
    int_T             nCoeffs     = nContStates + 1;
    int_T             activeBank  = ssGetIWork(S)[0];
    const real_T      *num        = ssGetRWork(S) + activeBank*(2*nCoeffs);
    const real_T      *den        = num + nCoeffs;
    InputRealPtrsType uPtrs       = ssGetInputPortRealSignalPtrs(S,0);

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

```
     */
    dx[0] = -den[1] * x[0] + *uPtrs[0];
    for (i = 1; i < nContStates; i++) {
        dx[i] = x[i-1];
        dx[0] -= den[i+1] * x[i];
    }

} /* end mdlDerivatives */



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


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

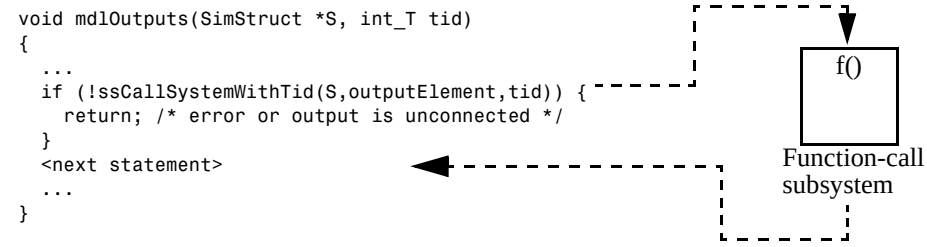
Function-Call Subsystems

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

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

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

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



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

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

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

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

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

2. Execute the subsystem in the appropriate `mdlOutputs` or `mdlUpdates` S-function routines. For example:

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

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

Function-call subsystems are a powerful modeling construct. You can configure Stateflow® blocks to execute function-call subsystems, thereby extending the capabilities and integration of state logic (Stateflow) with dataflow (Simulink). For more information on their use in Stateflow, see the Stateflow documentation.

The C MEX S-Function SimStruct

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

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

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

The following tables lists the macros that can be used by S-functions to access the SimStruct.

Table 3-5: General SimStruct Macros

Macro	Description
ssGetModelName(S)	For an S-function block, this is the name of the S-function MEX-file associated with the block (the term model in this context means an algorithm defined by your S-function). In the root SimStruct, this will be the name of the Simulink block diagram.
ssGetPath(S)	For S-function blocks, this is the full Simulink path to the S-function block. For the root SimStruct, this is equivalent to the model name. Within a C MEX S-function, in mdlInitializeSizes, if `strcmp(ssGetModelName(S),ssGetPath(S))==0` the S-function is being called from MATLAB and is not part of a simulation.

Table 3-5: General SimStruct Macros (Continued)

Macro	Description
`ssGetParentSS(S)`	This is typically not used in S-functions. It returns the parent `SimStruct` or `NULL` if the `S` is the root `SimStruct`.
`ssGetRootSS(s)`	This is typically not used in S-functions. It returns the root `SimStruct` of the tree of `SimStruct`'s.
`ssSetPlacementGroup (S,`*`groupName`*`)`	This is an advanced feature typically used for Real-Time Workshop device driver blocks. It is provided for S-functions that are either sources (i.e., no input ports) or sinks (i.e., no output ports). S-functions which share the same placement group name string will be placed adjacent to each other in block execution (i.e., the sorted list). The placement group should be set in `mdlInitializeSizes`.

Table 3-5: General SimStruct Macros (Continued)

Macro	Description
`ssSetOptions (S,`*`options`*`)`	Used in `mdlInitializeSizes` to set any of the following options. These options must be joined using the OR operator. For example: `ssSetOption(S, (SS_OPTION_EXCEPTION_FREE_CODE | SS_OPTION_DISCRETE_VALUED_OUTPUT));` `SS_OPTION_EXCEPTION_FREE_CODE` — If your S-function does not use `mexErrMsgTxt`, `mxCalloc`, or any other routines that can throw an exception when called, you can set this option for improved performance. `SS_OPTION_DISCRETE_VALUED_OUTPUT` — Specify this if your S-function has discrete valued outputs. This is checked when your S-function is placed within an algebraic loop. If your S-function has discrete valued outputs, then its outputs will not be assigned algebraic variables. `SS_OPTION_PLACE_ASAP` — Used to specify that your S-function should be placed as soon as possible. This is typically used by devices connecting to hardware. `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION` — Used to specify that the input to your S-function input ports can be either 1 or the size specified by the port, which is usually referred to as the block width.

Table 3-5: General SimStruct Macros (Continued)

Macro	Description
`ssSetOptions (S,`*`options`*`)` (continued)	`SS_OPTION_ASYNCHRONOUS` — This option applies only to S-functions that have 0 or 1 input ports and 1 output port. The output port must be configured to perform function calls on every element. If any of these requirements are not met, the `SS_OPTION_ASYNCHRONOUS` is ignored. Use this option when driving function-call subsystems that will be attached to interrupt service routines. `SS_OPTION_ASYNC_RATE_TRANSITION` — Use this when your S-function converts a signal from one rate to another rate. `SS_OPTION_RATE_TRANSITION` — Use this option when your S-function is behaving as a unit delay or a ZOH. This macro support these two operations only. It identifies a unit delay by the presence of `mdlUpdate`; if `mdlUpdate` is absent, the operation is taken to be ZOH.

Table 3-6: Error Handling and Status SimStruct Macros

Macros	Description

`ssSetErrorStatus (S,"string")`	For improved performance and error handling, your S-function should report errors using:
	`ssSetErrorStatus(S, "error message");`
	`return;`
	Be careful when using `ssSetErrorStatus` in your S-function. Error string must be persistent memory; it cannot be a local variable.

`ssGetSimMode(S)`	This macro can be used to determine the context in which your S-function is executing. Current simulation modes are: • `SS_SIMMODE_NORMAL` — Running a “normal” Simulink simulation • `SS_SIMMODE_SIZES_CALL_ONLY` — Block edit (evaluation) to obtain number of ports • `SS_SIMMODE_RTWGEN` — Generating code • `SS_SIMMODE_EXTERNAL` — External mode simulation
`ssGetSolverName(S)`	This returns a `char *` name of the solver.
`ssIsVariableStepSolver (S)`	Returns 1 if the solver being used is a variable step solver. This is useful when creating S-functions that have zero crossings and an inherited sample time.

Table 3-7: Input and Output Port Signal SimStruct Macros

Macro	Description
`ssSetNumInputPorts (S,nInputPorts)`	Used in `mdlInitializeSizes` to set to the number of input ports to a nonnegative integer. It should be invoked using: `if (!ssSetNumInputPorts(S,nInputPorts)) return;` where `ssSetNumInputPorts` returns a 0 if *`nInputPorts`* is negative or an error occurred while creating the ports. When this occurs, and you return out of your S-function, Simulink will display an error message.
`ssSetInputPortWidth (S,inputPortIdx,width)`	Used in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify a nonzero positive integer width or `DYNAMICALLY_SIZED` for each input port index starting at 0.
`ssSetInputPortDirectFeedThrough (S,inputPortIdx,dirFeed)`	Used in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the direct feedthrough (0 or 1) for each input port index. If not specified, the default direct feedthrough is 0. Setting direct feedthrough to 0 for an input port is equivalent to saying that the corresponding input port signal is not used in `mdlOutputs` or `mdlGetTimeOfNextVarHit`. If it is used, you may or may not see a delay of one simulation step in the input signal. This may cause the simulation solver to issue an error due to simulation inconsistencies.
`ssSetInputPortSampleTime (S,inputPortIdx,period)`	Used in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the sample time period as continuous or as a discrete value for each input port. Input port index numbers start at 0 and end at the total number of input ports minus 1. You should use this macro only if you have specified port-based sample times.

Table 3-7: Input and Output Port Signal SimStruct Macros (Continued)

Macro	Description
`ssSetInputPortOffsetTime (S,inputPortIdx,offset)`	Used in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the sample time offset for each input port index. You can use this macro in conjunction with `ssSetInputPortSampleTime` if you have specified port-based sample times for your S-function.
`ssSetInputPortOverWritable (S,inputPortIdx,value)`	Used in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify whether the input port is overwritable by an output port. The default is `value=0`, which means that the input port does not share memory with an output port. When `value=1`, the input port shares memory with an output port. Note that `ssSetInputPortReusable` and `ssSetOutputPortReusable` must both be set to 0, meaning that neither port involved can have global and persistent memory.

Table 3-7: Input and Output Port Signal SimStruct Macros (Continued)

Macro	Description
`ssSetInputPortReusable (S,inputPortIdx,value)`	Used in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify whether the input port memory buffer can be reused by other signals in the model. This macro can take on two values: • Off (`value=0`) — specifies that the input port is not reusable. This is the default. • On (`value=1`) — specifies that the input port is reusable. In Simulink, reusable signals share the same memory space. When this macro is turned on, the input port signal to the S-function may be reused by other signals in the model. This reuse results in less memory use during Simulink simulation and more efficiency in the Real-Time Workshop generated code. You must use caution when using this macro; you can safely turn it on only if the S-function reads its input port signal in its `mdlOutputs` routine and does not access this input port signal until the next call to `mdlOutputs`. When an S-functions's input port signal is reused, other signals in the model overwrite it prior to the execution of `mdlUpdate`, `mdlDerivatives`, or other run-time S-function routines. For example, if the S-function reads the input port signal in its `mdlUpdate` routine, or reads the input port signal in the `mdlOutputs` routine and expects this value to be persistent until the execution of its `mdlUpdate` routine, turning this attribute on is incorrect and will lead to erroneous results. The default setting, off, is safe. It prevents any reuse of the S-function input port signals, which means that the inport port signals have the same value in any run-time S-function routine during a single execution of the simulation loop.

Table 3-7: Input and Output Port Signal SimStruct Macros (Continued)

Macro	Description
`ssSetInputPortReusable (S,inputPortIdx,value)` (continued)	Note that this is a suggestion and not a requirement for the Simulink engine. If Simulink cannot resolve buffer reuse in local memory, it resets `value=0` and places the input port signals into global memory.
`ssSetNumOutputPorts (S,nOutputPorts)`	Used in `mdlInitializeSizes` to set to the number of output ports to a nonnegative integer. It should be invoked using: `if (!ssSetNumOutputPorts(S,nOutputPorts)) return;` where `ssSetNumOutputPorts` returns a 0 if *`nOutputPorts`* is negative or an error occurred while creating the ports. When this occurs, and you return out of your S-function, Simulink will display an error message.
`ssSetOutputPortWidth (S,outputPortIdx,width)`	Used in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify a nonzero positive integer width or `DYNAMICALLY_SIZED` for each output port index starting at 0.
`ssSetOutputPortSampleTime (S,outputPortIdx,period)`	Used in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify the sample time period as continuous or as a discrete value for each output port index. This should only be used if you have specified port-based sample times.
`ssSetOutputPortOffsetTime (S,outputPortIdx,offset)`	Used in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify the sample time offset value for each output port index. This should only be used if you have specified the S-function's sample times as port-based.

Table 3-7: Input and Output Port Signal SimStruct Macros (Continued)

Macro	Description
`ssSetOutputPortReusable (S,outputPortIdx,value)`	Used in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify whether output ports have a test point. This macro can take on two values: • Off (`value=0`) — specifies that the output port is not reusable. This is the default. • On (`value=1`) — specifies that the output port is reusable. In Simulink, reusable signals share the same memory space. When this macro is turned on, the output port signal to the S-function may be reused by other signals in the model. This reuse results in less memory use during Simulink simulation and more efficiency in the Real-Time Workshop generated code. When you mark an output port as reusable, your S-function must update the output once in `mdlOutputs`. It cannot expect the previous output value to be persistent. By default, the output port signals are not reusable. This forces Simulink's simulation engine (and the Real-Time Workshop) to allocate global memory for these output port signals. Hence this memory is only written to by your S-function and persists between model execution steps.
`ssGetNumInputPorts(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine how many input ports you have set.
`ssGetInputPortWidth (S,inputPortIdx)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the width of an input port.
`ssGetInputPortDirectFeedThrough (S,inputPortIdx)`	Can be used in any routine (except `mdlInitializeSizes`) to determine if an input port has direct feedthrough.

Table 3-7: Input and Output Port Signal SimStruct Macros (Continued)

Macro	Description

`ssGetInputPortRealSignalPtrs (S,`*`inputPortIdx`*`)`	Can be used in any simulation loop (see p. 3-16) S-function routine to access an input port signal. The input port index starts at 0 and ends at the number of input ports minus 1. This macro returns a pointer to an array of pointers to the `real_T` input signal elements. The length of the array of pointers is equal to the width of the input port. For example, to read all input port signals, use:
	`int_T i,j;`
	`int_T nInputPorts = ssGetNumInputPorts(S);`
	`for (i = 0; i < nInputPorts; i++) {`
	`  InputRealPtrsType uPtrs =`
	`  ssGetInputPortRealSignal(S,i);`
	`  int_T nu = ssGetInputPortWidth(S,i);`
	`  for (j = 0; j < nu; j++) {`
	`  `*`SomeFunctionToUseInputSignalElement`*`(*uPtrs[j]);`
	`  }`
	`}`

`ssGetInputPortSampleTime (S,`*`inputPortIdx`*`)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the sample time of an input port. This should only be used if you have specified the sample times as port-based.
`ssGetInputPortOffsetTime (S,`*`inputPortIdx`*`)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the offset time of an input port. This should only be used if you have specified the sample times as port-based.

Table 3-7: Input and Output Port Signal SimStruct Macros (Continued)

Macro	Description
`ssGetInputPortBufferDstPort (S,inputPortIdx)`	Can be used in any run-time (see p. 3-14) S-function routine to determine the index number of an output port when the specified input port has been overwritten by the output port. This can be used when you have specified the following: • The input port and some output port on an S-Function are *not* test points (`ssSetInputPortTestPoint` and `ssSetOutputPortTestPoint`) • The input port is over writable (`ssSetInputPortOverWritable`) If you have this set of conditions, then one of your S-functions's output ports (provided its test point has been turned off) may reuse the same buffer as the input port. If this happens, then after model initialization, the following macro returns the index of the output port that reuses the specified input port's buffer. If none of the S-Function's output ports reuse this input port buffer, then this macro returns `INVALID_PORT_IDX` (`= -1`).
`ssGetNumOutputPorts(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine how many output ports you have set.
`ssGetOutputPortWidth (S,outputPortIdx)`	Can be used 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.
`ssGetOutputPortSampleTime (S,outputPortIdx)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the sample time of an output port. This should only be used if you have specified port-based sample times.

Table 3-7: Input and Output Port Signal SimStruct Macros (Continued)

Macro	Description
`ssGetOutputPortOffsetTime (S,outputPortIdx)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the offset time of an output port. This should only be used if you have specified port-based sample times.

`ssGetOutputPortRealSignal (S,outputPortIdx)`	Can be used in any simulation loop routine, `mdlInitializeConditions`, or `mdlStart` to access an output port signal where the output port index starts at 0 and must be less than the number of output ports. This returns a contiguous `real_T` vector of length equal to the width of the output port. For 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();`
	`  }`
	`}`

Table 3-8: Parameter SimStruct Macros

Macros	Description
`ssSetNumSFcnParams (S,nSFcnParams)`	Used in `mdlInitializeSizes` to set the number of S-function parameters.
`ssGetSFcnParamsCount (S)`	Used in `mdlInitializeSizes` to get the number of parameters entered by in the S-function block dialog box.

Table 3-8: Parameter SimStruct Macros (Continued)

Macros	Description
`ssGetSFcnParam(S,index)`	Used in any routine to access a parameter entered by in the S-function block dialog box where *index* starts at 0 and is less than `ssGetSFcnParamsCount(S)`.
`ssSetSFcnParamNotTunable (S,index)`	Used 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 will improve efficiency and provide error handling in the event that an attempt is made to change the parameter.

Table 3-9: Sample Time SimStruct Macros

Macro	Description
`ssSetNumSampleTimes (S,nSampleTimes)`	Used in `mdlInitializeSizes` to set the number of sample times your S-function has. This must be a positive integer greater than 0.
`ssGetNumSampleTimes(S)`	Can be used in any routine (except `mdlIntializeSizes`) to get the number of sample times your S-function has.
`ssSetSampleTime (S,st_index, value)`	Used in `mdlInitializeSizes` to specify the "period" of the sample time where *st_index* starts at 0.
`ssSetOffsetTime (S,st_index,value)`	Used in `mdlInitializeSizes` to specify the "offset" of the sample time where *st_index* starts at 0.

Table 3-9: Sample Time SimStruct Macros (Continued)

Macro	Description
`ssIsSampleHit(S,st_index, tid)`	Used in `mdlOutputs` or `mdlUpdate` when your S-function has multiple sample times to determine what task your S-function is executing in. This should not be used in single rate S-functions or for the *`st_index`* corresponding to a continuous task.
`ssIsContinuousTask(S,tid)`	Used in `mdlOutputs` or `mdlUpdate` when your S-function has multiple sample times to determine if your S-function is executing in the continuous task. This should not be used in single rate S-functions, or if you did not register a continuous sample time.

Table 3-10: State and Work Vector SimStruct Macros

Macro	Description
`ssSetNumContStates (S,nContStates)`	Used in `mdlInitializeSizes` to specify the number of continuous states as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, then you can optionally specify the true (positive integer) width in `mdlSetWorkWidths`, otherwise the width to be used will be the width of the signal passing through the block. If your S-function has continuous states, then it needs to return the derivatives of the states in `mdlDerivatives` so that the solvers can integrate them. Continuous states will be logged when you have clicked the **States** checkbox on the **Workspace I/O** page of the **Simulation Parameters** dialog box.
`ssSetNumDiscStates (S,nDiscStates)`	Used in `mdlInitializeSizes` to specify the number of discrete states as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, then you can optionally specify the true (positive integer) width in `mdlSetWorkWidths`, otherwise the width to be used will be the width of the signal passing through the block. If your S-function has discrete states, then it should return the next discrete state (in place) in `mdlUpdate`. Discrete states will be logged when you have clicked the **States** checkbox on the **Workspace I/O** page of the **Simulation Parameters** dialog box.
`ssSetNumRWork(S,nRWork)`	Used 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`, then you can optionally specify the true (positive integer) width in `mdlSetWorkWidths`, otherwise the width to be used will be the width of the signal passing through the block.
`ssSetNumIWork(S,nIWork)`	Used 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`, then you can optionally specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise the width to be used will be the width of the signal passing through the block.

Table 3-10: State and Work Vector SimStruct Macros (Continued)

Macro	Description
`ssSetNumPWork(S,nPWork)`	Used 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`, then you can optionally specify the true (positive integer) width in `mdlSetWorkWidths`, otherwise the width to be used will be the width of the signal passing through the block.
`ssSetNumNonsampledZCs (S,nNonsampledZCs)`	Used in `mdlInitializeSizes` to specify the number of nonsampled zero crossings (`real_T`) as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, then you can optionally specify the true (positive integer) width in `mdlSetWorkWidths`, otherwise the width to be used will be the width of the signal passing through the block.
`ssSetNumModes(S,nModes)`	Used in `mdlInitializeSizes` to specify the number of modes (int_T's which are typically used with nonsampled zero crossings) as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, then you can optionally specify the true (positive integer) width in `mdlSetWorkWidths`, otherwise the width to be used will be the width of the signal passing through the block.
`ssGetNumContStates(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the number of continuous states your S-function is using.
`ssGetContStates(S)`	Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the `real_T` continuous state vector. This vector has length `ssGetNumContStates(S)`. Typically, this vector is initialized in `mdlInitializeConditions` and used in `mdlOutputs`.
`ssGetdX(S)`	This is used in `mdlDerivatives` to return the derivatives for your continuous states. This vector has length `ssGetNumContStates(S)`.

Table 3-10: State and Work Vector SimStruct Macros (Continued)

Macro	Description
`ssGetNumDiscStates(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the number of discrete states your S-function is using.
`ssGetRealDiscStates(S)`	Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the `real_T` discrete state vector. This vector has length `ssGetNumDiscStates(S)`. Typically, this vector is initialized in `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`.
`ssGetNumRWork(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the number of real work vector elements your S-function is using.
`ssGetRWork(S)`	Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the `real_T` work vector. This vector has length `ssGetNumRWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`.
`ssGetNumIWork(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the number of integer work vector elements your S-function is using.
`ssGetIWork(S)`	Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the `int_T` work vector. This vector has length `ssGetNumIWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`.
`ssGetNumPWork(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the number of pointer work vector elements your S-function is using.

Table 3-10: State and Work Vector SimStruct Macros (Continued)

Macro	Description
`ssGetPWork(S)`	Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the pointer (`void *`) work vector. This vector has length `ssGetNumPWork(S)`. Typically, this vector and its pointer contents are initialized in `mdlStart` or `mdlInitializeConditions`. Its contents are updated in `mdlUpdate`, and used in `mdlOutputs`.
`ssGetNumNonsampledZCs(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the number of nonsampled zero crossing vector elements your S-function is using.
`ssGetNonsampledZCs(S)`	This is used in `mdlZeroCrossings` to return the zero crossing signal values. The variable step solvers track the signals to locate where it crosses zero. The simulation time steps taken include these points. This vector has length `ssGetNumNonsampledZCs(S)`.
`ssGetNumModes(S)`	Can be used in any routine (except `mdlInitializeSizes`) to determine the number of mode vector elements your S-function is using.
`ssGetModeVector(S)`	Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the mode (`int_T`) work vector. This vector has length `ssGetNumModes(S)`. Typically, this vector is initialized in `mdlInitializeConditions` if the default value of zero 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. The mode vector may also be used in the `mdlZeroCrossings` routine to determine the current mode.

Table 3-11: Simulation Information SimStruct Macros

Macro	Description
`ssGetT(S)`	Used in `mdlOutputs` and `mdlUpdate` to get the "base" simulation time. If your S-function doesn't have a single rate continuous sample time, then this time may not be the "correct task time" and your S-function will run deterministically in a multitasking environment.
`ssGetTaskTime(S,st_index)`	Used in `mdlOutputs` or `mdlUpdate` to get the task time corresponding to a sample time index.
`ssGetTStart(S)`	Simulation start time.
`ssGetTFinal(S)`	Simulation stop time.
`ssIsMinorTimeStep(S)`	Used in `mdlOutputs` to determine if the routine is being called in a minor time step.
`ssIsMajorTimeStep(S)`	Used in `mdlOutputs` to determine if the routine is being called in a major time step.
`ssSetStopRequested(S,val)`	Can be called in any simulation loop routine to specify that the simulation should stop at the end of the current time step.
`ssSetSolverNeedsReset(S)`	Used to inform the solvers that the equations which are being integrated have changed. This macro differs slightly in format from the other macros in that you don't specify a value; this was by design so that invoking it always requests a reset.

Table 3-12: Function-Call SimStruct Macros

Macro	Description
`ssSetCallSystemOutput(S,index)`	Used in `mdlInitializeSampleTimes` to specify that the a the specified output port 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)`.

`ssCallSystemWithTid(S,index,tid)`	Used in `mdlOutputs` to execute a function-call subsystem connected to the S-function. The invoking syntax is:
	`if (!ssCallSystemWithTid(S,index, tid)) {`
	`  /* Error occurred which will be reported by`
	`  Simulink */`
	`  return;`
	`}`

Converting Level 1 C MEX S-Functions to Level 2

Level 2 S-functions were introduced with Simulink 2.2. Level 1 S-functions refer to S-functions that were written to work with Simulink 2.1 and previous releases. Level 1 S-functions are compatible with Simulink 2.2; 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`. 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 `mdlIntializeSizes`, 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 */
```

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`, then update it to the following form

```
#define MDL_INITIALIZE_CONDITIONS
static void mdlInitializeConditions(SimStruct *S)
{
}
```

  otherwise, delete the function.

  - The continuous states are accessed using `ssGetContStates`. The `ssGetX` macro has been removed.
  - The discrete states are accessed 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 may 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 into Level-2 S-functions, you must use:

  - `ssGetInputPortRealSignalPtrs` to access inputs.
  - `ssGetOutputPortSignal` to access the outputs.
  - `ssGetContStates` or `ssGetRealDiscStates` to access the states.

- The `mdlUpdate` function prototype has been 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`, then 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`, then update it to this form:

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

  otherwise, delete the function.

- Replace all obsolete SimStruct macros. See Table 3-13 for a complete list of obsolete macros.

- When converting level 1 S-functions to level 2 S-functions, you should build your S-functions with full (i.e., highest) warning levels. For example, if you have gcc on a UNIX system, use these options with the `mex` utility:

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

  If your system has Lint, use this code:

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

  On a PC, to use the highest warning levels, you must create a project file inside of 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).

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

Table 3-13: Obsolete SimStruct Macros

Obsolete Macro	Replace With
`ssGetU(S), ssGetUPtrs(S)`	`ssGetInputPortSignalPtrs(S,port)`
`ssGetY(S)`	`ssGetOutputPortRealSignal(S,port)`
`ssGetX(S)`	`ssGetContStates(S), ssGetRealDiscStates(S)`
`ssGetStatus(S)`	Normally not used, but `ssGetErrorStatus(S)` is available.
`ssSetStatus(S,msg)`	`ssSetErrorStatus(S,msg)`
`ssGetSizes(S)`	Specific call the desired 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)`	`ssGetSampleTime(S,sti,t)`
`ssGetOffsetTimeEvent(S,sti)`	`ssGetOffsetTime(S,sti)`
`ssSetOffsetTimeEvent(S,sti,t)`	`ssSetOffsetTime(S,sti,t)`
`ssIsSampleHitEvent(S,sti,tid)`	`ssIsSampleHit(S,sti,tid)`
`ssGetNumInputArgs(S)`	`ssGetNumSFcnParams(S)`
`ssSetNumInputArgs(S, numInputArgs)`	`ssSetNumSFcnParams(S,numInputArgs)`
`ssGetNumArgs(S)`	`ssGetSFcnParamsCount(S)`
`ssGetArg(S,argNum)`	`ssGetSFcnParam(S,argNum)`
`ssGetNumInputs`	`ssGetNumInputPorts(S)` and `ssGetInputPortWidth(S,port)`
`ssSetNumInputs`	`ssSetNumInputPorts(S,nInputPorts)` and `ssSetInputPortWidth(S,port,val)`

Table 3-13: Obsolete SimStruct Macros (Continued)

Obsolete Macro	Replace With
`ssGetNumOutputs`	`ssGetNumOutputPorts(S)` and `ssGetOutputPortWidth(S,port)`
`ssSetNumOutputs`	`ssSetNumOutputPorts(S,nOutputPorts)` and `ssSetOutputPortWidth(S,port,val)`

4

Guidelines for Writing C MEX S-Functions

Introduction

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

This chapter assumes that you understand these concepts:

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

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

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

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

Classes of Problems Solved by S-Functions

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

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

S-functions and S-function routines form an application program interface (API) that allows you to implement generic algorithms in the Simulink

environment with a great deal of flexibility. This flexibility cannot always be maintained when you use S-functions with the Real-Time Workshop. For example, it is not possible to access the MATLAB workspace from an S-function that is used with the Real-Time Workshop. However, using the techniques presented in this chapter, you can create S-functions for most applications that work with the generated code from the Real-Time Workshop.

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

Types of S-Functions

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

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

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

   or said another way:

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

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

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

1 Noninlined S-function

2 Wrapper S-function

3 Fully inlined S-function

Noninlined S-Functions

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

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

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

Wrapper S-Functions

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

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

Fully Inlined S-Functions

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

Basic Files Required for Implementation

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

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

For S-functions to work correctly in the Simulink environment, a certain amount of overhead code is necessary. When the Real-Time Workshop generates code from models that contain S-functions (without *`sfunction`*`.tlc`

files), it embeds some of this overhead code in the generated C code. If you want to optimize your real-time code and eliminate some of the overhead code, you must *inline* (or embed) your S-functions. This involves writing a TLC (`sfunction.tlc`) file that directs the Real-Time Workshop to eliminate all overhead code from the generated code. The Target Language Compiler, which is part of the Real-Time Workshop, processes `sfunction.tlc` files to define how to inline your S-function algorithm in the generated code.

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

Noninlined S-Functions

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

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

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

Writing Wrapper S-Functions

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

The MEX S-Function Wrapper

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

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

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

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

- interpretive in nature in Simulink (i.e., highly-parameterized by operating modes)
- heavily optimized in the Real-Time Workshop (i.e., no extra tests to decide what mode the code is operating in)

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

This figure illustrates the wrapper S-function concept:

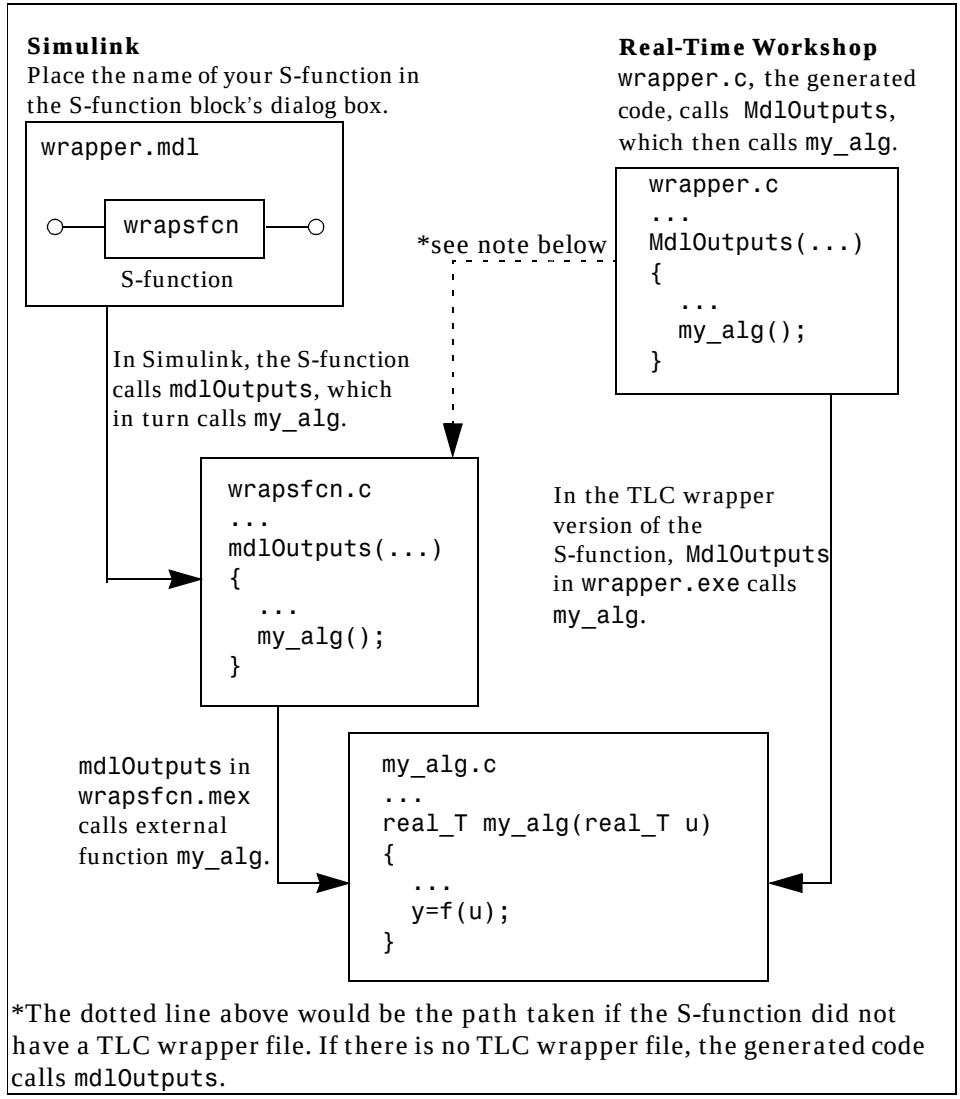


Figure 4-1: How S-Functions Interface with Hand-Written Code

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

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

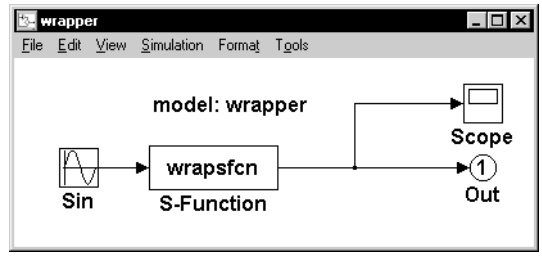


Figure 4-1: An Example Model That Includes an S-Function Wrapper

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

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

extern real_T my_alg(real_T u);

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

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

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

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

    ssSetNumSampleTimes( S, 1);
}
```

Declare `my_alg` as `extern`.

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


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

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

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

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

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

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

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

The TLC S-Function Wrapper

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

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

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

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

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

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

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

#include <math.h>
#include <string.h>
#include “wrapper.h”
#include “wrapper.prm”

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

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

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

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

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

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

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

#include "wrapper.reg"

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

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

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

How to Inline

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

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

This call has a significant amount of computational overhead associated with it. First, Simulink creates a SimStruct data structure for the S-function block. Second, the Real-Time Workshop constructs a call through a function pointer to execute MdlOutputs, and then MdlOutputs calls my_alg. By inlining the call

to your C algorithm (`my_alg`), you can eliminate both the `SimStruct` and the extra function call, thereby improving the efficiency and reducing the size of the generated code.

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

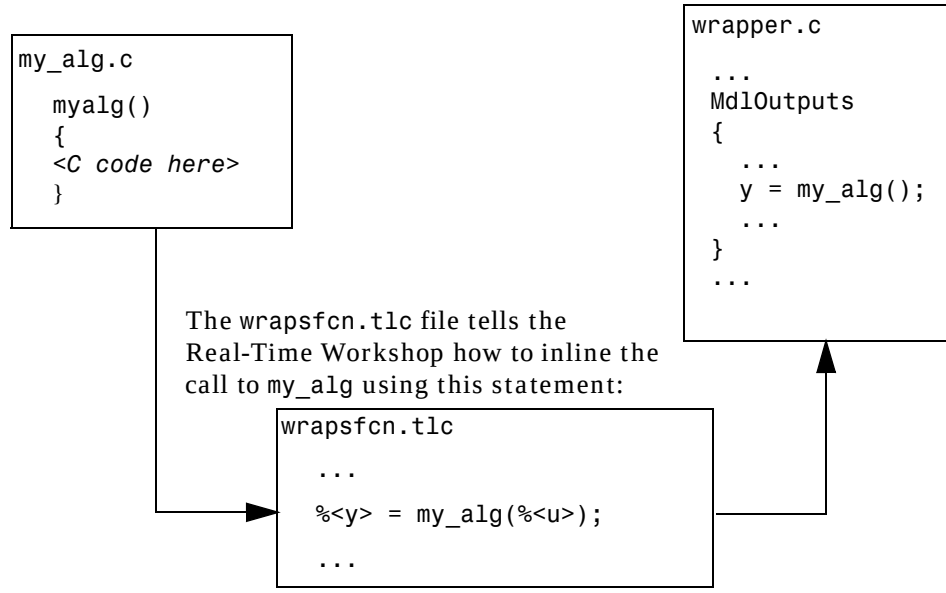


Figure 4-2: Inlining an Algorithm by Using a TLC File

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

placing calls to your S-function in the generated code. This is the `wrapsfcn.tlc` file that inlines `wrapsfcn.c`:

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

%implements “wrapsfcn” “C”

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

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

%endfunction %% Outputs
```

This line is placed in `wrapper.h`.

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

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

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

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

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

The Inlined Code

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

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

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

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

Inlined call to the function `my_alg`.

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

Fully Inlined S-Functions

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

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

use:

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

This is the code produced in `MdlOutputs`:

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

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

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

This is the explicit embedding of the algorithm.

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

Multiport S-Function Example

A more advanced multiport inlined S-function example exists in *`matlabroot`*`/simulink/src/sfun_multiport.c` and *`matlabroot`*`/toolbox/simulink/blocks/sfun_multiport.tlc`. This S-function demonstrates how to create a fully inlined TLC file for an S-function

that contains multiple ports. You may find that looking at this example will aid in the understanding of fully inlined TLC files.

Fully Inlined S-Function with the mdlRTW Routine

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

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

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

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

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

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

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

The Direct-Index Lookup Table Algorithm

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

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

The parameters to the S-function are:

```
XData, YData, XEvenlySpaced
```

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

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

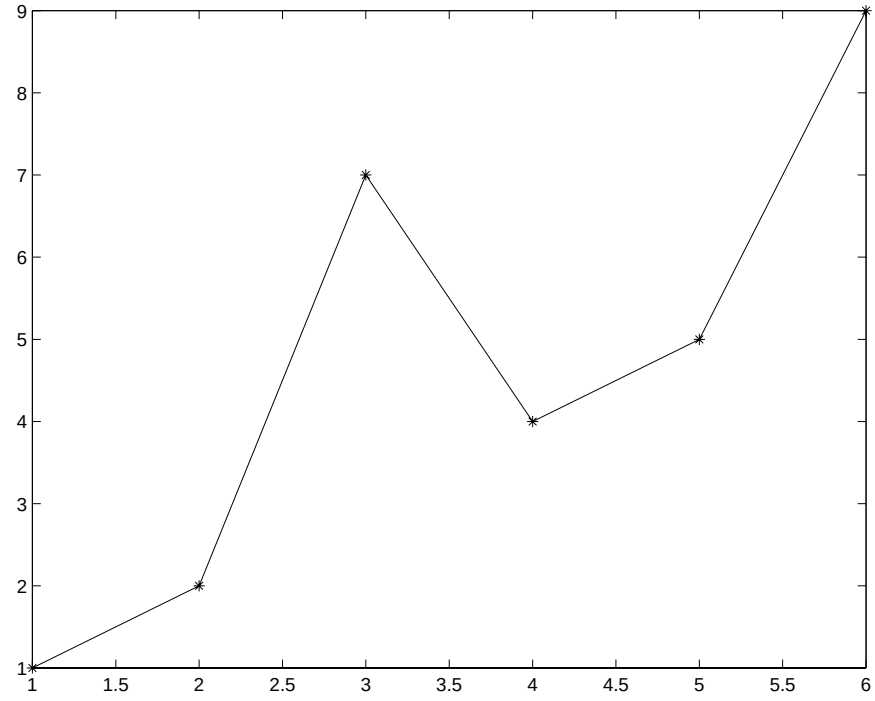


Figure 4-3: Typical Output from a Lookup Table Example

The Direct-Index Lookup Table Example

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

Implementation of the direct-index algorithm with inlined TLC file requires the S-function main module, `sfun_directlook.c` (see page 4–27) and a corresponding `lookup_index.c` module (see page 4–36). The `lookup_index.c` module contains the `GetDirectLookupIndex` routine that is used to locate the

index in the XData for the current x input value when the XData is unevenly spaced. The GetDirectLookupIndex routine is called from both the S-function and the generated code. Here the example uses the wrapper concept for sharing C code between Simulink MEX-files and the generated code.

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

The inlined TLC file, which performs either a wrapper call or embeds the optimal C code, is sfun_directlook.tlc (see page 4–38).

Error Handling

In this example, the mdlCheckParameters routine on page 4–29 verifies that:

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

Note that the mdlInitilizeSizes routine explicitly calls mdlCheckParameters after it has verified the number of parameters passed to the S-function are correct. After Simulink calls mdlInitializeSizes, it will then call mdlCheckParameters whenever you change the parameters or there is a need to re-evaluate them.

User Data Caching

The mdlStart routine on page 4–32 illustrates how to cache information that does not change during the simulation (or while the generated code is executing). The example caches the value of the XDataEvenlySpaced parameter in UserData, a field of the SimStruct. The

```
ssSetSFcnParamNotTunable(S, XDATAEVENLYSPACED_PIDX);
```

line in mdlInitializeSizes tells Simulink to disallow changes to the XDataEvenlySpaced parameter. During execution, mdlOutputs accesses the value of XDataEvenlySpaced from the UserData rather than calling the

mxGetPr MATLAB API function. This results in a slight increase in performance.

mdlRTW Usage

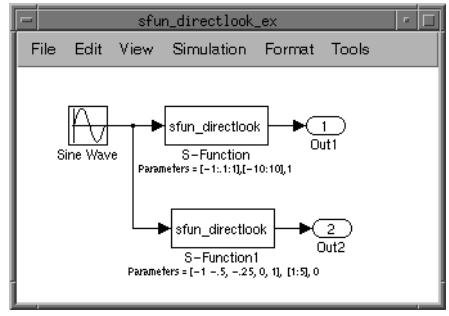
The Real-Time Workshop calls the mdlRTW routine while it (the Real-Time Workshop) generates the *model*.rtw file. You can add information to the *model*.rtw file about the mode in which your S-function block is operating to produce optimal code for your Simulink model.

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

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

Example Model

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



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

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

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

The generated code for the lookup table example model is

```
<Generated header for sfun_directlook_ex model>

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

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

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

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

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

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

  /* Outport Block: <Root>/Out1 */
```

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

```
  rtY.Out1 = rtb_buffer2;

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

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

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

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

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

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

#include “sfun_directlook_ex.reg”

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

matlabroot/simulink/src/sfun_directlook.c

```
/*
 * File    :  sfun_directlook.c
 * Abstract:
 *
 *      Direct 1-D lookup. Here we are trying to compute an approximate
 *      solution, p(x) to an unknown function f(x) at x=x0, given data point
 *      pairs (x,y) in the form of a x data vector and a y data vector. For a
 *      given data pair (say the i'th pair), we have y_i = f(x_i). It is
 *      assumed that the x data values are monotonically increasing.  If the
 *      x0 is outside of the range of the x data vector, then the first or
 *      last point will be returned.
 *
 *      This function returns the "nearest" y0 point for a given x0. No
 *      interpolation is performed.
 *
 *      The S-function parameters are:
 *        XData              - double vector
 *        YData              - double vector
 *        XDataEvenlySpacing - double scalar 0 (false) or 1 (true)
 *        The third parameter cannot be changed during simulation.
 *
 *      To build:
 *        mex sfun_directlook.c lookup_index.c
 *
 * Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 * $Revision: 1.3 $
 */

#define S_FUNCTION_NAME  sfun_directlook
#define S_FUNCTION_LEVEL 2


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

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

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

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

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

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

typedef struct SFcnCache_tag {
    boolean_T evenlySpaced;
} SFcnCache;

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


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


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

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

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

```
          }

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


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


#define MDL_CHECK_PARAMETERS           /* Change to #undef to remove function */
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
/* Function: mdlCheckParameters =============================================
 * Abstract:
 *    This routine will be called after mdlInitializeSizes, whenever
 *    parameters change or get re-evaluated. The purpose of this routine is
 *    to verify that the new parameter settings are correct.
 *
 *    You should add a call to this routine from mdlInitalizeSizes
 *    to check the parameters. After setting your sizes elements, you should:
 *       if (ssGetSFcnParamsCount(S) == ssGetNumSFcnParams(S)) {
 *           mdlCheckParameters(S);
 *       }
 */
static void mdlCheckParameters(SimStruct *S)
{

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

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

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

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

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

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

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

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

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

        if (badSpacing) {
            ssSetErrorStatus(S,"X-vector must be an evenly spaced "
                             "strictly monotonically increasing vector");
            return;
        }
```

```
        } else {     /* XData is 'unevenly-spaced' */
            for (i = 1; i < numEl; i++) {
                if (xData[i] <= xData[i-1]) {
                    ssSetErrorStatus(S,"X-vector must be a strictly "
                                       "monotonically increasing vector");
                    return;
                }
            }
        }
    }
}
#endif /* MDL_CHECK_PARAMETERS */



/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, NUM_PARAMS);  /* Number of expected parameters */

    /*
     * Check parameters passed in, providing the correct number was specified
     * in the S-function dialog box. If an incorrect number of parameters
     * was specified, Simulink will detect the error since ssGetNumSFcnParams
     * and ssGetSFcnParamsCount will differ.
     *   ssGetNumSFcnParams   - This sets the number of parameters your
     *                          S-function expects.
     *   ssGetSFcnParamsCount - This is the number of parameters entered by
     *                          the user in the Simulink S-function dialog box.
     */
#if defined(MATLAB_MEX_FILE)
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink */
    }
#endif

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

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

    ssSetInputPortTestPoint(S, 0, FALSE);
```

```
    ssSetInputPortOverWritable(S, 0, TRUE);

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

    ssSetOutputPortTestPoint(S, 0, FALSE);

    ssSetNumSampleTimes(S, 1);

    ssSetSFcnParamNotTunable(S, XDATAEVENLYSPACED_PIDX);

    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);

} /* mdlInitializeSizes */


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



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

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

    ssSetUserData(S, cache);

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

```
        cache->evenlySpaced = FALSE;
    }

}
#endif /*  MDL_START */



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

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

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

```
} /* end mdlOutputs */



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



#define MDL_RTW                         /* Change to #undef to remove function */
#if defined(MDL_RTW) && (defined(MATLAB_MEX_FILE) || defined(NRT))
/* Function: mdlRTW ===========================================================
 * Abstract:
 *    This function is called when the Real-Time Workshop is generating the
 *    model.rtw file. In this routine, you can call the following functions
 *    which add fields to the model.rtw file.
 *
 *    Important! Since this S-function has this mdlRTW routine, it must have
 *    a correSponding .tlc file to work with the Real-Time Workshop. You will find
 *    the sfun_directlook.tlc in the same directory as sfun_directlook.dll.
 */
static void mdlRTW(SimStruct *S)
{
    /*
     * Write out the [X,Y] data as parameters, i.e., these values can be
     * changed during execution.
     */
    {
        real_T *xData = mxGetPr(XVECT(S));
        int_T  xLen   = mxGetNumberOfElements(XVECT(S));
        real_T *yData = mxGetPr(YVECT(S));
        int_T  yLen   = mxGetNumberOfElements(YVECT(S));

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

```
        boolean_T even = (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0);

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


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

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


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

matlabroot/simulink/src/lookup_index.c

```
/* File    : lookup_index.c
 * Abstract:
 *
 *     Contains a routine used by the S-function sfun_directlookup.c to
 *     compute the index in a vector for a given data value.
 *
 *  Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 *  $Revision: 1.3 $
 */
#include “tmwtypes.h”

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

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

    /*
     * We have: x[bottom] < u < x[top], onward
     * with search for the appropriate index ...
```

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

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

matlabroot/toolbox/simulink/blocks/sfun_directlook.tlc

```
%% File    : sfun_directlook.tlc
%% Abstract:
%%      Level-2 S-function sfun_directlook block target file.
%%      It is using direct lookup algorithm without interpolation.
%%
%% Copyright (c) 1994-1998 by The MathWorks, Inc. All Rights Reserved.
%% $Revision: 1.3 $


%implements “sfun_directlook” “C”

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

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

  %<LibCacheFunctionPrototype(buffer)>

%endfunction

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

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

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

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

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

```
        %endif
      %endroll
    %endif
  }

%endfunction

%% EOF: sfun_directlook.tlc
```

Index

O

P

R

S

T

V

W