
C S-Function Techniques

You can find the complete documentation of the various S-function techniques in *Writing S-Functions* by The MathWorks. This PDF contains an excerpt of aspects that are especially important when using S-functions in a dSPACE environment.

The following standard S-function techniques are described here:

Using templates

Since all S-functions use the same set of individual routines (C S-function callback methods), it is recommended that you start your programming with an S-function template. Refer to *How to Use an S-Function Template* on page 4.

Size information

Each S-function must specify its size information. Refer to *Size Information* on page 5.

Accessing inports and outports

Usually you want to connect the S-Function blocks of your model to other blocks, which means that you need to access the inports and outports of the S-Function blocks within the S-function's source code. Refer to *Accessing the Inports and Outports of an S-Function Block* on page 7.

Accessing inports in mdlOutputs()

Outports of your S-function might depend directly on one or more inports of the S-function, which is called a direct feedthrough. Refer to *Accessing Inport Signals in mdlOutputs()* on page 9.

Using persistent memory

It might be necessary to use the results from one calculation step in the next one or the results of one S-function method in another. Refer to *How to Use Persistent Memory in S-Functions* on page 10.

Multirate S-functions

You can create multirate S-functions, where parts of the code are executed less often than others. Or you might want to avoid the `mdlOutputs()` method being called multiple times in each simulation step when using higher-order integration algorithms for the model. Refer to *How to Use Sample Times in an S-Function* on page 12.

Enabled subsystems

Enabled subsystems can hold or reset states when they are enabled, which also affects the behavior of S-functions. Refer to *S-Functions Within Enabled Subsystems* on page 14.

Generating the MEX file

For each S-function you must generate the corresponding MEX DLL file (MATLAB executable). See *How to Generate the Corresponding MEX DLL File* on page 15.

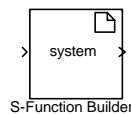
Placing the S-Function block

To use an S-function in your model, you have to add and configure an S-Function block to the model. Refer to *How to Use S-Functions in a Simulink Model* on page 17.

Related Topics

Using the S-Function Builder block

As an alternative to a handcoded S-function, you can also use the S-Function Builder block to implement simple C coded S-functions. This is a Simulink block that builds an S-function from specifications and C code that you supply. The block also serves as a wrapper for the generated S-function. The block is available from the User-Defined Functions library and is shown below.



For details on implementing S-functions via this block, refer to *Writing S-Functions* by The MathWorks.

How to Use an S-Function Template

Since all S-functions use the same set of individual routines (C S-function callback methods), it is recommended that you start your programming with an S-function template.

- If you want to create a new S-function (such as in the level control example), take the S-function template `sfuntmpl_basic.c` file from the `<MATLAB_ROOT>\simulink\src\` folder.
- If you want to modify an RTI-specific S-function, take the desired original S-function C file from the `%DSPACE_ROOT%\MATLAB\RTI<XXXX>\SFcn\` folder.



The original S-functions are restored and updated during any new installation. So before you modify any template, you should copy it to the working folder that holds your model and rename the template. It is then available for all models located in that folder.

To get an S-function template

- 1 Copy the desired S-function template to the working folder that holds your Simulink model.
- 2 Choose a name for the S-function and rename the file accordingly. Enter the S-function name (without `.c` extension) at the macro definition `S_FUNCTION_NAME` in the C file.

When you configure the S-Function block for the Simulink model, make sure to enter the S-function name (without `.c` extension) in the Block Parameters dialog.

Size Information

The size information is a very vital piece of information for an S-function, since it defines the number of inputs, outputs, states, and sample times. Simulink and Real-Time Workshop require this information to check whether the S-Function block has the correct number of ports, which sample time is used, etc. Therefore, you have to specify this information in the `mdlInitializeSizes()` and `mdlInitializeSampleTimes()` source code methods.



The following code excerpt shows the size information for an S-function that reads one Boolean input in the `mdlOutputs()` method. It has no parameters, no states, no outputs. It has only one sample time and is not executed in the minor time steps of higher-order integration algorithms.

```

/* Function: mdlInitializeSizes ===== */
static void mdlInitializeSizes(SimStruct *S)
{
/* This S-function has no parameters. */
ssSetNumSFcnParams(S, 0); /* No. of expected parameters */
if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
    return;
}
/* Set the number of states to zero. */
ssSetNumContStates(S, 0);
ssSetNumDiscStates(S, 0);
/* Set one input port. */
if (!ssSetNumInputPorts(S, 1)) return;
/* Set width of input port to one. */
ssSetInputPortWidth(S, 0, 1);
/* Input is accessed in mdlOutputs() routine,
   therefore set direct feedthrough flag. */
ssSetInputPortDirectFeedThrough(S, 0, 1);
/* Set the input port data type to boolean. */
ssSetInputPortDataType(S, 0, SS_BOOLEAN);
/* We have no output ports, therefore we must not
   call ssSetOutputPortWidth()! */
if (!ssSetNumOutputPorts(S, 0)) return;
/* This S-function has one sample time. */
ssSetNumSampleTimes(S, 1);
/* No work vectors and other special features are used. */
ssSetNumRWork(S, 0);

```

```
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);
    /* No additional options are used. */
    ssSetOptions(S, 0);
}
/* Function: mdlInitializeSampleTimes ===== */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    /* Use inherited sample time. */
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    /* Ensure that this S-function will not execute in
       intermediate integration steps. */
    ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
}
```

Accessing the Inports and Outports of an S-Function Block

An S-Function block can have several inports and outports, each of which can be scalar or vectorized and have a different data type.

The following topics provide a detailed description of how you can access inports. The technique for accessing outports is very similar (using `ssGetOutputPortRealSignal` and `ssGetOutputPortSignal`).



The following code excerpt shows how to access all the inports and their individual signals using the `ssGetInputPortRealSignalPtrs` access macro. For example, you could use this code fragment in the `mdlOutputs()` S-function method. This example assumes that all inports have the double data type.

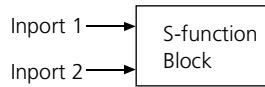
```
int_T idxPort, idxSignal;
int_T nInputPorts = ssGetNumInputPorts(S);
for (idxPort = 0; idxPort < nInputPorts; idxPort++)
{
    real_T input;
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, idxPort);
    int_T nPortWidth = ssGetInputPortWidth(S, idxPort);
    for (idxSignal = 0; idxSignal < nPortWidth; idxSignal++)
    {
        input = *uPtrs[idxSignal];
        SomeFunctionToUseInputSignalElement(input);
    }
}
```



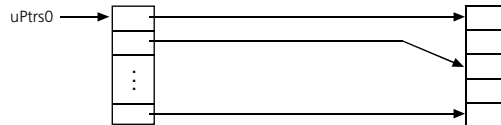
The pointer for each inport must be fetched separately via the `ssGetInputPortRealSignalPtrs(S, i)` macro. This macro is only valid for real-typed input signals. If you want to process a signal of another type, you have to use `ssGetInputPortSignalPtrs`.

If the port is vectorized, the retrieved pointer points to the first element of the vector. To reach the subsequent signals, you can simply increment the pointer as done with `*uPtrs[idxSignal]` in the example.

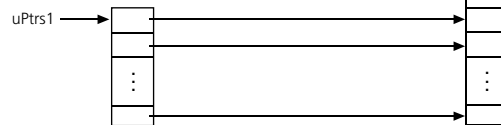
As you can see from the following illustration, the input variables of the individual inports are not located one after another in memory. Therefore you cannot reach the signals of different ports by just incrementing the pointer to the first inport.



To Access Inport 1:
InputRealPtrsType uPtrs0 =
ssGetInputPortRealSignalPtrs(S,0)



To Access Inport 2:
InputRealPtrsType uPtrs1 =
ssGetInputPortRealSignalPtrs(S,1)



See also *Accessing Signals Using Pointers* in *Writing S-Functions* by The Mathworks.

Accessing Inport Signals in `mdlOutputs()`

If your S-function has inports and outputs, you typically calculate the output signals within the `mdlOutputs()` method using at least one of the input signals. In terms of S-functions, such an inport has a direct feedthrough, which must be specified via the following entry in the `mdlInitializeSizes()` method:

```
void ssSetInputPortDirectFeedThrough(SimStruct *S, int_T port, int_T dirFeed)
```



If the direct feedthrough flag is incorrectly set for an inport, the result of the calculation is unpredictable and you can encounter runtime errors that are very hard to find.



Suppose you want to set the direct feedthrough flag for inport 2:

```
ssSetInputPortDirectFeedThrough(S, 2, 1)
```

How to Use Persistent Memory in S-Functions

Suppose you want to use the results from one calculation step in the next one, which means that you need to retain the values of variables between the successive calls of the S-function methods. If you want to use several copies (instances) of the same S-function in your model, you must ensure that the S-function is reentrant, so that the execution of one instance does not influence the execution of another. In this case, you can use work vectors for data storage. Work vectors provide the necessary instance-specific storage for block variables and therefore grant reentrancy for the S-function.

Work vectors must be initialized in `mdlInitializeConditions()` or `mdlStart()`, they are updated in `mdlUpdate()` (discrete states), and used in `mdlOutputs()`.

To use work vectors in an S-function

- 1 Define the size and data types of the required work vectors in the `mdlInitializeSizes()` S-function method via the macros:

```
ssSetNumRWork(S, <NUM_RWORK>); /* doubles typed*/
ssSetNumIWork(S, <NUM_IWORK>); /* integer typed*/
ssSetNumPWork(S, <NUM_PWORK>); /* pointer typed*/
```

- 2 Use the following macros to access the pointers that reference the work vectors:

```
ssGetRWork(S); /* doubles typed*/
ssGetIWork(S); /* integer typed*/
ssGetPWork(S); /* pointer typed*/
```



To observe the work vectors, for example, via ControlDesk, they must be available in the system description file. The easiest solution is to create additional outputs in your S-function because outputs are automatically generated into the system description file.



You should not use global or static variables to store S-function data because global variables are used by all instances of an S-function and therefore such S-functions are not reentrant.

Suppose your model contains two instances (a and b) of an S-function that uses a set of global variables. If instance (a) writes data to the global variables and then (b) is called and writes data, (a) gets the wrong data the next time it is called.

See also *Writing S-Functions* by The MathWorks.

How to Use Sample Times in an S-Function

If desired, you can specify different sample times for the internal processes of the S-Function block.

Port based You can define individual sample times for the inports and outports of the S-Function block. For information, refer to the documentation of The MathWorks.

Block based If no sample times are specified for inports and outports, they are updated with the fastest sample rate of the S-function. This is most commonly used.

To assign a specific sample rate to a code block

- 1 Include the following lines in the `mdlInitializeSampleTimes()` S-function method for each sample time you want to use:

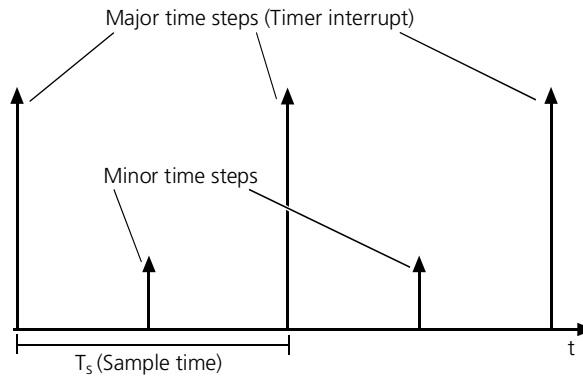
```
ssSetSampleTime(S, <SAMPLE_TIME_IDX>, <SAMPLE_TIME>);  
ssSetOffsetTime(S, <SAMPLE_TIME_IDX>, <SAMPLE_TIME_OFFSET>);
```

- 2 Identify the code block you want to assign to a specific sample rate and enclose it in an `if` statement using `ssIsSampleHit(...)`, for example:

```
if(ssIsSampleHit(S, sample_time_index, tid))  
{  
    /* code block assigned to the sample time with  
    sample_time_index */  
}
```

Higher-order integration

If you select a higher-order, fixed-step numerical integration algorithm for your model, i.e. ode2 ... ode5, the `mdlOutputs()` method is called not only for the major time steps but also for the minor time steps to improve the precision of the simulation results. This is shown in the following diagram.



When you use an S-function to control hardware, you might need to suppress the calls to `mdlOutputs()` at the minor time steps because certain hardware has a limited maximum sample rate. For example, an A/D converter needs some time for each conversion process.

To exclude code blocks from the minor time steps

- Set the `FIXED_IN_MINOR_STEP_OFFSET` flag in the `ssSetOffsetTime` method, for example:

```
ssSetOffsetTime(S, 1, FIXED_IN_MINOR_STEP_OFFSET);
```

As a result, the code blocks assigned to the sample time with ID "1" are excluded from the minor time steps.

S-Functions Within Enabled Subsystems

Enabled subsystems provide two options for the **States when enabling** setting, which you can select via the Block Parameters dialog of the Enable block:

Hold The states of blocks within the subsystem remain constant when the subsystem is disabled. When the subsystem is reenabled, the held states are used for simulation.

Any contained S-function is initialized once at simulation start (`mdlStart()` and `mdlInitializeConditions()` are executed).

Reset The states of blocks within the subsystem are reset to their initial values. When the subsystem is reenabled, the simulation continues with the initial values.

Any contained S-function is initialized at simulation start (`mdlStart()` and `mdlInitializeConditions()` are executed) and each time the subsystem becomes enabled (`mdlInitializeConditions()` is executed).

Make sure to place the initializations in the correct S-function method:

mdlStart() Use this method for initializations that need to be carried out only once, for example, when initializing I/O devices.

mdlInitializeCondition() Use this method for initializations that must be carried out each time the subsystem becomes enabled, for example, initializing variables.



If an I/O device is initialized more than once, for example, if you place the initialization in `mdlInitializeConditions()` by mistake, the behavior of the device can become unpredictable. Therefore, use `mdlStart()` for initializations that must occur only during simulation start.

How to Generate the Corresponding MEX DLL File

In addition to the source C file of an S-function, Simulink and Real-Time Workshop require a compiled version of the S-function: the MEX DLL file. If you perform a Simulink simulation of your model, Simulink uses the MEX DLL file to calculate the S-function. If you generate real-time code, Real-Time Workshop needs the MEX DLL file to read the size information of the S-function.

You can generate the MEX DLL file for an S-function via MATLAB's `mex` command. However, MATLAB needs to be set up properly before you can use the `mex` command.

To set up MATLAB's MEX command

- 1 Install an appropriate C compiler if this has not already been done. For example, you can use the LCC compiler, which is part of the MATLAB installation.

For further information on valid C compilers, refer to *External Interfaces/API* by The MathWorks.

- 2 Change to the MATLAB prompt and type the `mex -setup` command. Then follow the instructions on the screen.

When MATLAB is ready to compile MEX DLL files, you can generate the MEX DLL file for an S-function as follows:

To generate the corresponding MEX DLL file

- Type `mex <source>.c -v` at the MATLAB prompt. Replace `<source>` with the file name of your C source file.

The `-v` option makes the MEX compiler display the options used, and issue any warnings.



- If you created a new S-function, you always have to generate a MEX DLL file. However, if you modified an existing S-function, you might not need to recompile the MEX DLL file. You can reuse the old MEX DLL file if you want to use the S-function only for real-time simulation and you did not change its size information or sample time.

- If you want to hold your MEX DLL files in a separate folder, you have to add the folder to the MATLAB path (via the `addpath` command or the path tool) and RTI's make search path (via the `<model>_usr.mk` file).

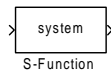
Refer to *Calling C and Fortran Programs from MATLAB* ([apiext.pdf](#)) by The MathWorks for information on the available options for the `mex` command.

How to Use S-Functions in a Simulink Model

Since the S-Function block has no function on its own you have to handcode the desired functionality in C.

To include S-functions in a Simulink model

- 1 Place the **S-Function** block from the Simulink library in your model and open its Block Parameters dialog.



If you want to include an S-function that was generated from a Simulink model, you have to use the RTW S-Function block. After you have built a model using the RTW S-Function, Real-Time Workshop opens a new model and inserts a correctly configured RTW S-Function block.



The Texas Instruments compiler Ver. 4.70 cannot compile a model that contains S-functions generated by Real-Time Workshop. If you want to use them, you need to use Ver. 5.11.

- 2 In the **S-function name** edit field, enter the name (without the `.c` file name extension) you want to assign to the desired S-function. The underlying C file needs to have the same name. If the S-function requires any parameters, you can specify them in this dialog as well.



Once the MEX DLL file is available for the S-function, the S-Function block automatically adapts the correct number of ports.

