/\*

\* File: sfuntmpl\_doc.c

\* Abstract:

\* A 'C' template for a level 2 S-function.

\* 运行在2级的S函数

\* See sfuntmpl\_basic.c

\* for a basic C-MEX template file that uses the

\* most common methods.

\* sfuntmpl\_basic.c常用的基本C-mex模板

\* Copyright 1990-2013 The MathWorks, Inc.

\*/

/\*

\* You must specify the S\_FUNCTION\_NAME as the name of your S-function.

必须在文件开头设定函数名

\*/

#define S\_FUNCTION\_NAME your\_sfunction\_name\_here

#define S\_FUNCTION\_LEVEL 2

/\*

\* Need to include simstruc.h for the definition of the SimStruct and

\* its associated macro definitions.

\*

\* The following headers are included by matlabroot/simulink/include/simstruc.h

\* when compiling as a MEX file:

\* 编译MEX文件时包含以下头文件

\* matlabroot/extern/include/tmwtypes.h - General 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

\*

\* The following headers are included by matlabroot/simulink/include/simstruc.h

\* when compiling your S-function with RTW:

\* 在RTW下编译时包含以下头文件

\* matlabroot/extern/include/tmwtypes.h - General types, e.g. real\_T

\* matlabroot/rtw/c/libsrc/rt\_matrx.h - Macros for MATLAB API routines

\*

\*/

#include "simstruc.h"

/\* Error handling

关于错误处理

\* --------------

\*

\* You should use the following technique to report errors encountered within

\* an S-function:

\*

\* ssSetErrorStatus(S,"error encountered due to ...");

\* return;

\*

\* Note that the 2nd 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];"}

其中要使用static

\* 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. 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 setup exception handlers prior to each S-function invocation.

\* This introduces overhead into simulation.

\*

\* If you do not call mexErrMsgTxt or any other routines that cause exceptions,

\* then you should use 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 the

\* SS\_OPTION\_EXCEPTION\_FREE\_CODE option. If your S-function generates

\* an exception when this option is set, unpredictable results will occur.

\*

\* Exception free code refers to code which never "long jumps". Your S-function

\* is not exception free if it contains any routine which 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.

\* Use of mxCalloc may cause unpredictable problems in the event of a memory

\* allocation error since mxCalloc will long jump. If memory allocation is

\* needed, you should use the stdlib.h calloc routine directly and perform

\* your own error handling.

\*

\* All mex\* routines have the potential of long jumping (i.e. throwing an

\* exception). In addition several mx\* routines have the potential of

\* long jumping. To avoid any difficulties, only the routines which get

\* a pointer or determine the size of parameters should be used. For example

\* the following will never throw an exception: mxGetPr, mxGetData,

\* mxGetNumberOfDimensions, mxGetM, mxGetN, mxGetNumberOfElements.

\*

\* If all of your "run-time" methods within your S-function are exception

\* free, then you can use the option:

\* ssSetOptions(S, SS\_OPTION\_RUNTIME\_EXCEPTION\_FREE\_CODE);

\* The other methods in your S-function need not be exception free. The

\* run-time methods include any of the following:

\* mdlGetTimeOfNextVarHit, mdlOutputs, mdlUpdate, and mdlDerivatives

\*

\* Warnings & Printf's

\* -------------------

\* You can use ssWarning(S,msg) to display a warning.

\* - When the S-function is compiled via mex for use with Simulink,

\* ssWarning equates to mexWarnMsgTxt.

\* - When the S-function is used with Real-Time Workshop,

\* ssWarning(S,msg) equates to

\* printf("Warning: in block '%s', '%s'\n", ssGetPath(S),msg);

\* if the target has stdio facilities, otherwise it becomes a comment and

\* is disabled.

\*

\* You can use ssPrintf(fmt, ...) to print a message.

\* - When the S-function is compiled via mex for use with Simulink,

\* ssPrintf equates to mexPrintf.

\* - When the S-function is used with Real-Time Workshop,

\* ssPrintf equates to printf, if the target has stdio facilities,

\* otherwise it becomes a call to a empty function (rtPrintfNoOp).

\* - In the case of Real-Time Workshop which may or may not have stdio

\* facilities, to generate the most efficient code use:

\* #if defined(SS\_STDIO\_AVAILABLE)

\* ssPrintf("my message ...");

\* #endif

\* - You can also use this technique to do other standard I/O related items,

\* such as:

\* #if defined(SS\_STDIO\_AVAILABLE)

\* if ((fp=fopen(file,"w")) == NULL) {

\* ssSetErrorStatus(S,"open failed");

\* return;

\* }

\* ...

\* #endif

\*/

/\*====================\*

\* S-function methods \*

\*====================\*/

/\*

\* Level 2 S-function methods

\* --------------------------

\* Notation: "=>" indicates method is required.

\* [method] indicates method is optional.

\*

\* Note, many of the methods below are only available for use in level 2

\* C-MEX S-functions.

\*

\* Model Initialization in Simulink 初始化阶段

\* --------------------------------

\*=> mdlInitializeSizes 必需 - Initialize SimStruct sizes array

\*

\* [mdlSetInputPortFrameData] - Optional method. Check and set input and

\* output port frame data attributes.

\*

\* NOTE: An S-function cannot use mdlSetInput(Output)PortWidth and

\* mdlSetInput(Output)PortDimensionInfo at the same time. It can use

\* either a width or dimension method, but not both.

\*

\* [mdlSetInputPortWidth] - Optional method. Check and set input and

\* optionally other port widths.

\* [mdlSetOutputPortWidth] - Optional method. Check and set output

\* and optionally other port widths.

\*

\* [mdlSetInputPortDimensionInfo]

\* - Optional method. Check and set input and

\* optionally other port dimensions.

\* [mdlSetOutputPortDimensionInfo]

\* - Optional method. Check and set output

\* and optionally other port dimensions.

\* [mdlSetDefaultPortDimensionInfo]

\* - Optional method. Set dimensions of all

\* input and output ports that have unknown

\* dimensions.

\*

\* [mdlSetInputPortSampleTime] - Optional method. Check and set input

\* port sample time and optionally other port

\* sample times.

\* [mdlSetOutputPortSampleTime]- Optional method. Check and set output

\* port sample time and optionally other port

\* sample times.

\*=> mdlInitializeSampleTimes 必需 - Initialize sample times and optionally

\* function-call connections.

\*

\* [mdlSetInputPortDataType] - Optional method. Check and set input port

\* data type. See SS\_DOUBLE to SS\_BOOEAN in

\* simstruc\_types.h for built-in data types.

\* [mdlSetOutputPortDataType] - Optional method. Check and set output port

\* data type. See SS\_DOUBLE to SS\_BOOLEAN in

\* simstruc\_types.h for built-in data types.

\* [mdlSetDefaultPortDataTypes] - Optional method. Set data types of all

\* dynamically typed input and output ports.

\*

\* [mdlInputPortComplexSignal] - Optional method. Check and set input

\* port complexity attribute (COMPLEX\_YES,

\* COMPLEX\_NO).

\* [mdlOutputPortComplexSignal] - Optional method. Check and set output

\* port complexity attribute (COMPLEX\_YES,

\* COMPLEX\_NO).

\* [mdlSetDefaultPortComplexSignals]

\* - Optional method. Set complex signal flags

\* of all input and output ports who

\* have their complex signals set to

\* COMPLEX\_INHERITED (dynamic complexity).

\*

\* [mdlSetWorkWidths] - Optional method. Set the state, iwork,

\* rwork, pwork, dwork, etc sizes.

\*

\* [mdlStart] 可选 - Optional method. Perform actions such

\* as allocating memory and attaching to pwork

\* elements.

\*

\* [mdlInitializeConditions] - Initialize model parameters (usually

\* states). Will not be called if your

\* S-function does not have an initialize

\* conditions method.

\*

\* ['constant' mdlOutputs] - Execute blocks with constant sample

\* times. These are only executed once

\* here.

\*

\* [mdlSetSimState] - Optional method. Load the complete simulation

\* state for this block, which is called when

\* starting the simulation from an initial

\* simulation state and this s-function has set

\* its ssSetSimStateCompliance to

\* USE\_CUSTOM\_SIM\_STATE. See also mdlGetSimState

\*

\* Model simulation loop in Simulink 循环仿真阶段

\* ---------------------------------

\* [mdlCheckParameters] - Optional method. Will be called at

\* any time during the simulation loop when

\* parameters change.

\* SimulationLoop:

\* [mdlProcessParameters] - Optional method. Called during

\* simulation after parameters have been

\* changed and verified to be okay by

\* mdlCheckParameters. The processing is

\* done at the "top" of the simulation loop

\* when it is safe to process the changed

\* parameters.

\* [mdlGetTimeOfNextVarHit] 用于变步长 - Optional method. If your S-function

\* has a variable step sample time, then

\* this method will be called.

\* [mdlInitializeConditions]- Optional method. Only called if your

\* S-function resides in an enabled

\* subsystem configured to reset states,

\* and the subsystem has just enabled.

\* => mdlOutputs 必需 - Major output call (usually updates

\* output signals).

\* [mdlUpdate] - Update the discrete states, etc.

\*

\* Integration (Minor time step) 积分步骤

\* [mdlDerivatives] - Compute the derivatives.

\* Do

\* [mdlOutputs]

\* [mdlDerivatives]

\* EndDo - number of iterations depends on solver

\* Do

\* [mdlOutputs]

\* [mdlZeroCrossings]

\* EndDo - number of iterations depends on zero crossings signals

\* EndIntegration

\*

\* EndSimulationLoop

\*

\* [mdlGetSimState] - Optional method. Called to get the complete simulation

\* state for this block if the model is configured to

\* save its final simulation state and this

\* S-Function has set its ssSetSimStateCompliance to

\* USE\_CUSTOM\_SIM\_STATE. See also mdlSetSimState

\*

\* => mdlTerminate 必需 - End of model housekeeping - free memory, etc.

\*

\* Model initialization for code generation (rtwgen)

\* -------------------------------------------------

\* <Initialization. See "Model Initialization in Simulink" above>

\*

\* [mdlRTW] - Optional method. Only called when

\* generating code to add information to the

\* model.rtw file which is used by the

\* Real-Time Workshop.

\*

\* mdlTerminate - End of model housekeeping - free memory,

\* etc.

\*

\* Noninlined S-function execution in Real-Time Workshop

在RTW内执行的非内联函数

\* -----------------------------------------------------

\* 1) The results of most initialization methods are 'compiled' into

\* the generated code and many methods are not called.

\* 2) Noninlined S-functions are limited in several ways, for example

\* parameter must be real (non-complex) double vectors or strings.

非内联S函数的参数必需为实数、double型向量或字符串

More capability is provided via the Target Language Compiler. See the Target Language Compiler Reference Guide.

\* => mdlInitializeSizes 初始化SimStruct - Initialize SimStruct sizes array

\* => mdlInitializeSampleTimes - Initialize sample times and optionally

\* 初始化采样时间 function-call connections.

\* [mdlInitializeConditions] - Initialize model parameters (usually

\* states). Will not be called if your

\* S-function does not have an initialize

\* conditions method.

\* [mdlStart] - Optional method. Perform actions such

\* 在这一步初始化寄存器 as allocating memory and attaching to pwork

\* elements.

\* ExecutionLoop:

\* => mdlOutputs - Major output call (usually updates

\* 输出 output signals).

\* [mdlUpdate] - Update the discrete states, etc.

\*

\* Integration (Minor time step)

\* [mdlDerivatives] - Compute the derivatives.

\* Do

\* [mdlOutputs]

\* [mdlDerivatives]

\* EndDo - number of iterations depends on solver

\* Do

\* [mdlOutputs]

\* [mdlZeroCrossings]

\* EndDo - number of iterations depends on zero crossings signals

\* EndExecutionLoop

\* mdlTerminate - End of model housekeeping - free memory,

\* etc.

\*/

/\*====================================================================\*

\* Parameter handling methods. These methods are not supported by RTW \*

\*====================================================================\*/

参数处理方法。以下方法不支持RTW

#define MDL\_CHECK\_PARAMETERS /\* Change to #undef to remove function \*/

#if defined(MDL\_CHECK\_PARAMETERS) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlCheckParameters =============================================

\* Abstract:

\* mdlCheckParameters verifies new parameter settings whenever parameter

\* 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.

\*

\* This method can be called at any point after mdlInitializeSizes.

\* You should add a call to this method from mdlInitalizeSizes

\* to check the parameters. After setting the number of parameters

\* you expect in your S-function via ssSetNumSFcnParams(S,n), you should:

\* #if defined(MATLAB\_MEX\_FILE)

\* if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {

\* mdlCheckParameters(S);

\* if (ssGetErrorStatus(S) != NULL) return;

\* } else {

\* return; Simulink will report a parameter mismatch error

\* }

\* #endif

\*

\* When a Simulation is running, changes to S-function parameters can

\* occur either at the start of a simulation step, or during a

\* simulation step. When changes to S-function parameters occur during

\* a simulation step, this method is called twice, for the same

\* parameter changes. 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 that you cannot access the

\* work, state, input, output, etc. vectors in this method. This

\* method should only be used to validate the parameters. Processing

\* of the parameters should be done in mdlProcessParameters.

\*

\* See matlabroot/simulink/src/sfun\_errhdl.c for an example.

\*/

static void mdlCheckParameters(SimStruct \*S)

{

}

#endif /\* MDL\_CHECK\_PARAMETERS \*/

#define MDL\_PROCESS\_PARAMETERS /\* Change to #undef to remove function \*/

#if defined(MDL\_PROCESS\_PARAMETERS) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlProcessParameters ===========================================

\* Abstract:

\* This method will be called after mdlCheckParameters, whenever

\* parameters change or get re-evaluated. The purpose of this method is

\* to process the newly changed parameters. For example "caching" the

\* parameter changes in the work vectors. Note this method is not

\* called when it is used with the Real-Time Workshop. Therefore,

\* if you use this method in an S-function which is being used with the

\* Real-Time Workshop, you must write your S-function such that it doesn't

\* rely on this method. This can be done by inlining your S-function

\* via the Target Language Compiler.

\*/

static void mdlProcessParameters(SimStruct \*S)

{

}

#endif /\* MDL\_PROCESS\_PARAMETERS \*/

/\*=====================================\*

\* Configuration and execution methods \*

\*=====================================\*/

/\* Function: mdlInitializeSizes ===============================================

\* Abstract:

\* The sizes information is used by Simulink to determine the S-function

\* block's characteristics (number of inputs, outputs, states, etc.).

\*

\* Direct Feedthough:

\* 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 or

\* mdlGetTimeOfNextVarHit() function. Setting this to 0 is akin to making

\* a promise that "u" will not be used in the mdlOutput or

\* mdlGetTimeOfNextVarHit() function. If you break the promise, then

\* unpredictable results will occur.

\*

\* It is very common for S-function authors to write incorrect S-functions

\* when configuring the S-function direct feedthrough flag. We often find

\* that S-function authors are confused about what the correct setting for

\* the direct feedthrough flag should be. Part of the confusion is because

\* the term direct feedthrough is misleading. To reduce the confusion, you

\* can think of the direct feedthrough setting as a 'needs input'

\* setting. Specifically, if the S-function access an input signal in

\* either mdlOutputs() or mdlGetTimeOfNextVarHit(), then the direct

\* feedthrough flag must be set to 1 (true).

\*

\* For example, if a level 2 C-MEX S-function uses:

\* ssGetInputPortSignal(S,inputPortIndex)

\* in its mdlOutputs() or mdlGetTimeOfNextVarHit() methods, then the

\* S-function is required to set the direct feedthrough flag to true in

\* its mdlInitializeSizes() method:

\* ssSetInputPortDirectFeedThrough(S, inputPortIdx, 1);

\*

\* If your S-Function uses ssGetInputPortSignal() in mdlOutputs() or

\* mdlGetTimeOfNextVarHit() and fails to specify

\* the direct feedthrough, or specifes it wrongly using

\* ssSetInputPortDirectFeedThrough(S, inputPortIdx, 0);

\* then the S-Function is incorreclty written and Simulink will not

\* operate correctly and may crash.

\*

\* When you compile your S-function with debugging, e.g.,

\* mex -g sfunction\_name.c

\* your S-function will be instrumented such that an incorrect access to

\* an input signal will generate a diagnostic message.

\*

\* 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)

{

int\_T nInputPorts = 1; /\* number of input ports \*/

int\_T nOutputPorts = 1; /\* number of output ports \*/

int\_T needsInput = 1; /\* direct feed through \*/

int\_T inputPortIdx = 0;

int\_T outputPortIdx = 0;

ssSetNumSFcnParams(S, 0); /\* Number of expected parameters \*/

if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {

/\*

\* If the the number of expected input parameters is not equal

\* to the number of parameters entered in the dialog box return.

\* Simulink will generate an error indicating that there is a

\* parameter mismatch.

\*/

return;

}

/\*

\* Configure tunability of parameters. By default, all parameters are

\* tunable (changeable) during simulation. If there are parameters that

\* cannot change during simulation, such as any parameters that would change

\* the number of ports on the block, the sample time of the block, or the

\* data type of a signal, mark these as non-tunable using a call like this:

\*

\* ssSetSFcnParamTunable(S, 0, 0);

\*

\* which sets parameter 0 to be non-tunable (0).

\*

\*/

/\* Register the number and type of states the S-Function uses \*/

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.

\*/

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

/\*

\* Set input port dimensions for each input port index starting at 0.

\* The following options summarize different ways for setting the input

\* port dimensions.

\*

\* (1) If the input port dimensions are unknown, use

\* ssSetInputPortDimensionInfo(S, inputPortIdx, DYNAMIC\_DIMENSION))

\*

\* (2) If the input signal is an unoriented vector, and the input port

\* width is w, use

\* ssSetInputPortVectorDimension(S, inputPortIdx, w)

\* w (or width) can be DYNAMICALLY\_SIZED or greater than 0.

\* This is equivalent to ssSetInputPortWidth(S, inputPortIdx, w).

\*

\* (3) If the input signal is a matrix of dimension mxn, use

\* ssSetInputPortMatrixDimensions(S, inputPortIdx, m, n)

\* m and n can be DYNAMICALLY\_SIZED or greater than zero.

\*

\* (4) Otherwise use:

\* ssSetInputPortDimensionInfo(S, inputPortIdx, dimsInfo)

\* This function can be used to fully or partially initialize the port

\* dimensions. dimsInfo is a structure containing width, number of

\* dimensions, and dimensions of the port.

\*/

if(!ssSetInputPortDimensionInfo(S, inputPortIdx, DYNAMIC\_DIMENSION)) return;

/\*

\* Set direct feedthrough flag (1=yes, 0=no).

\* A port has direct feedthrough if the input is used in either

\* the mdlOutputs or mdlGetTimeOfNextVarHit functions.

\* See sfuntmpl\_directfeed.txt.

\*/

ssSetInputPortDirectFeedThrough(S, inputPortIdx, needsInput);

/\*

\* Configure the output ports. First set the number of output ports.

\*/

if (!ssSetNumOutputPorts(S, nOutputPorts)) return;

/\*

\* Set output port dimensions for each output port index starting at 0.

\* See comments for setting input port dimensions.

\*/

if(!ssSetOutputPortDimensionInfo(S,outputPortIdx,DYNAMIC\_DIMENSION)) return;

/\*

\* 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 method. 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 \*/

/\* Specify the sim state compliance to be same as a built-in block \*/

/\* see sfun\_simstate.c for example of other possible settings \*/

ssSetSimStateCompliance(S, USE\_DEFAULT\_SIM\_STATE);

/\*

\* All options have the form SS\_OPTION\_<name> and are documented in

\* matlabroot/simulink/include/simstruc.h. The options should be

\* bitwise or'd together as in

\* ssSetOptions(S, (SS\_OPTION\_name1 | SS\_OPTION\_name2))

\*/

ssSetOptions( S, 0); /\* general options (SS\_OPTION\_xx) \*/

} /\* end mdlInitializeSizes \*/

#define MDL\_SET\_INPUT\_PORT\_FRAME\_DATA /\* Change to #undef to remove function \*/

#if defined(MDL\_SET\_INPUT\_PORT\_FRAME\_DATA) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetInputPortFrameData ========================================

\* Abstract:

\* This method is called with the candidate frame setting (FRAME\_YES, or

\* FRAME\_NO) for an input port. If the proposed setting is acceptable,

\* the method should go ahead and set the actual frame data setting using

\* ssSetInputPortFrameData(S,portIndex,frameData). If

\* the setting is unacceptable an error should generated via

\* ssSetErrorStatus. Note that any other dynamic frame input or

\* output ports whose frame data setting are implicitly defined by virtue

\* of knowing the frame data setting of the given port can also have their

\* frame data settings set via calls to ssSetInputPortFrameData and

\* ssSetOutputPortFrameData.

\*/

static void mdlSetInputPortFrameData(SimStruct \*S,

int portIndex,

Frame\_T frameData)

{

} /\* end mdlSetInputPortFrameData \*/

#endif /\* MDL\_SET\_INPUT\_PORT\_FRAME\_DATA \*/

#define MDL\_SET\_INPUT\_PORT\_WIDTH /\* Change to #undef to remove function \*/

#if defined(MDL\_SET\_INPUT\_PORT\_WIDTH) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetInputPortWidth ===========================================

\* Abstract:

\* This method is called with the candidate width for a dynamically

\* sized port. If the proposed width is acceptable, the method should

\* go ahead and set the actual port width using ssSetInputPortWidth. If

\* the size is unacceptable an error should generated via

\* ssSetErrorStatus. Note that any other dynamically sized input or

\* output ports whose widths are implicitly defined by virtue of knowing

\* the width of the given port can also have their widths set via calls

\* to ssSetInputPortWidth or ssSetOutputPortWidth.

\*/

static void mdlSetInputPortWidth(SimStruct \*S, int portIndex, int width)

{

} /\* end mdlSetInputPortWidth \*/

#endif /\* MDL\_SET\_INPUT\_PORT\_WIDTH \*/

#define MDL\_SET\_OUTPUT\_PORT\_WIDTH /\* Change to #undef to remove function \*/

#if defined(MDL\_SET\_OUTPUT\_PORT\_WIDTH) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetOutputPortWidth ==========================================

\* Abstract:

\* This method is called with the candidate width for a dynamically

\* sized port. If the proposed width is acceptable, the method should

\* go ahead and set the actual port width using ssSetOutputPortWidth. If

\* the size is unacceptable an error should generated via

\* ssSetErrorStatus. Note that any other dynamically sized input or

\* output ports whose widths are implicitly defined by virtue of knowing

\* the width of the given port can also have their widths set via calls

\* to ssSetInputPortWidth or ssSetOutputPortWidth.

\*/

static void mdlSetOutputPortWidth(SimStruct \*S, int portIndex, int width)

{

} /\* end mdlSetOutputPortWidth \*/

#endif /\* MDL\_SET\_OUTPUT\_PORT\_WIDTH \*/

#undef MDL\_SET\_INPUT\_PORT\_DIMENSION\_INFO /\* Change to #define to add function \*/

#if defined(MDL\_SET\_INPUT\_PORT\_DIMENSION\_INFO) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetInputPortDimensionInfo ====================================

\* Abstract:

\* This method is called with the candidate dimensions for an input port

\* with unknown dimensions. If the proposed dimensions are acceptable, the

\* method should go ahead and set the actual port dimensions.

\* If they are unacceptable an error should be generated via

\* ssSetErrorStatus.

\* Note that any other input or output ports whose dimensions are

\* implicitly defined by virtue of knowing the dimensions of the given

\* port can also have their dimensions set.

\*

\* See matlabroot/simulink/src/sfun\_matadd.c for an example.

\*/

static void mdlSetInputPortDimensionInfo(SimStruct \*S,

int\_T portIndex,

const DimsInfo\_T \*dimsInfo)

{

} /\* mdlSetInputPortDimensionInfo \*/

#endif /\* MDL\_SET\_INPUT\_PORT\_DIMENSION\_INFO \*/

#undef MDL\_SET\_OUTPUT\_PORT\_DIMENSION\_INFO /\* Change to #define to add function\*/

#if defined(MDL\_SET\_OUTPUT\_PORT\_DIMENSION\_INFO) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetOutputPortDimensionInfo ===================================

\* Abstract:

\* This method is called with the candidate dimensions for an output port

\* with unknown dimensions. If the proposed dimensions are acceptable, the

\* method should go ahead and set the actual port dimensions.

\* If they are unacceptable an error should be generated via

\* ssSetErrorStatus.

\* Note that any other input or output ports whose dimensions are

\* implicitly defined by virtue of knowing the dimensions of the given

\* port can also have their dimensions set.

\*

\* See matlabroot/simulink/src/sfun\_matadd.c for an example.

\*/

static void mdlSetOutputPortDimensionInfo(SimStruct \*S,

int\_T portIndex,

const DimsInfo\_T \*dimsInfo)

{

} /\* mdlSetOutputPortDimensionInfo \*/

#endif /\* MDL\_SET\_OUTPUT\_PORT\_DIMENSION\_INFO \*/

#undef MDL\_SET\_DEFAULT\_PORT\_DIMENSION\_INFO /\* Change to #define to add fcn \*/

#if defined(MDL\_SET\_DEFAULT\_PORT\_DIMENSION\_INFO) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetDefaultPortDimensionInfo ==================================

\* Abstract:

\* This method is called when there is not enough information in your

\* model to uniquely determine the port dimensionality of signals

\* entering or leaving your block. When this occurs, Simulink's

\* dimension propagation engine calls this method to ask you to set

\* your S-functions default dimensions for any input and output ports

\* that are dynamically sized.

\*

\* If you do not provide this method and you have dynamically sized ports

\* where Simulink does not have enough information to propagate the

\* dimensionality to your S-function, then Simulink will set these unknown

\* ports to the 'block width' which is determined by examining any known

\* ports. If there are no known ports, the width will be set to 1.

\*

\* See matlabroot/simulink/src/sfun\_matadd.c for an example.

\*/

static void mdlSetDefaultPortDimensionInfo(SimStruct \*S)

{

} /\* mdlSetDefaultPortDimensionInfo \*/

#endif /\* MDL\_SET\_DEFAULT\_PORT\_DIMENSION\_INFO \*/

#define MDL\_SET\_INPUT\_PORT\_SAMPLE\_TIME

#if defined(MDL\_SET\_INPUT\_PORT\_SAMPLE\_TIME) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetInputPortSampleTime =======================================

\* Abstract:

\* This method is called with the candidate sample time for an inherited

\* sample time input port. If the proposed sample time is acceptable, the

\* method 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

\* virtue of knowing the sample time of the given port can also have

\* their sample times set via calls to ssSetInputPortSampleTime or

\* ssSetOutputPortSampleTime.

\*

\* When inherited port based sample times are specified, we are guaranteed

\* that the sample time will be one of the following:

\* [sampleTime, offsetTime]

\* continuous [0.0 , 0.0 ]

\* discrete [period , offset ] where 0.0 < period < inf

\* 0.0 <= offset < period

\* Constant, triggered, and variable step sample times will not be

\* propagated to S-functions with port based sample times.

\*

\* Generally the mdlSetInputPortSampleTime or mdlSetOutputPortSampleTime

\* is called once with the input port sample time. However, there can be

\* cases where this function will be called more than once. This happens

\* when the simulation engine is converting continuous sample times to

\* continuous but fixed in minor steps sample times. When this occurs, the

\* original values of the sample times specified in mdlInitializeSizes

\* will be restored before calling this method again.

\*

\* The final sample time specified at the port may be different (but

\* equivalent to) from what was specified in this method. This occurs

\* when:

\* o) Using a fixed step solver and the port has a continuous but fixed

\* in minor step sample time. In this case the sample time will

\* be converted to the fundamental sample time for the model.

\* o) We are adjusting sample times for numerical correctness. For

\* example [0.2499999999999, 0] is converted to [0.25, 0].

\* S-functions are not explicitly notified of "converted" sample times.

\* They can examine the final sample times in mdlInitializeSampleTimes.

\*/

static void mdlSetInputPortSampleTime(SimStruct \*S,

int\_T portIdx,

real\_T sampleTime,

real\_T offsetTime)

{

} /\* end mdlSetInputPortSampleTime \*/

#endif /\* MDL\_SET\_INPUT\_PORT\_SAMPLE\_TIME \*/

#define MDL\_SET\_OUTPUT\_PORT\_SAMPLE\_TIME

#if defined(MDL\_SET\_OUTPUT\_PORT\_SAMPLE\_TIME) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetOutputPortSampleTime ======================================

\* Abstract:

\* This method is called with the candidate sample time for an inherited

\* sample time output port. If the proposed sample time is acceptable, the

\* method should go ahead and set the actual port sample time using

\* ssSetOutputPortSampleTime. 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

\* virtue of knowing the sample time of the given port can also have

\* their sample times set via 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, we call this method in succession

\* for all inherited output port signals.

\*

\* See mdlSetInputPortSampleTimes for more information about when this

\* method is called.

\*/

static void mdlSetOutputPortSampleTime(SimStruct \*S,

int\_T portIdx,

real\_T sampleTime,

real\_T offsetTime)

{

} /\* end mdlSetOutputPortSampleTime \*/

#endif /\* MDL\_SET\_OUTPUT\_PORT\_SAMPLE\_TIME \*/

/\* Function: mdlInitializeSampleTimes =========================================

\* Abstract:

\*

\* This function is used to specify the sample time(s) for your S-function.

\* You must register the same number of sample times as specified in

\* ssSetNumSampleTimes. If you specify that you have no sample times, then

\* the S-function is assumed to have one inherited sample time.

\*

\* The sample times are specified as pairs "[sample\_time, offset\_time]"

\* via the following 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:

\*

\* o A continuous function that changes during minor integration steps

\* should register the [CONTINUOUS\_SAMPLE\_TIME, 0.0] sample time.

\* o A continuous function that does not change during minor integration

\* steps should register the

\* [CONTINUOUS\_SAMPLE\_TIME, FIXED\_IN\_MINOR\_STEP\_OFFSET]

\* sample time.

\* o 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

\* o 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.

\* Note, the VARIABLE\_SAMPLE\_TIME can be used with variable step

\* solvers only.

\* o Discrete blocks which can operate in triggered subsystems. For your

\* block to operate correctly in a triggered subsystem or a periodic

\* system it must register [INHERITED\_SAMPLE\_TIME, 0.0]. In a triggered

\* subsystem after sample times have been propagated throughout the

\* block diagram, the assigned sample time to the block will be

\* [INHERITED\_SAMPLE\_TIME, INHERITED\_SAMPLE\_TIME]. Typically discrete

\* blocks which can be periodic or reside within triggered subsystems

\* need to register the inherited sample time and the option

\* SS\_OPTION\_DISALLOW\_CONSTANT\_SAMPLE\_TIME. Then in mdlSetWorkWidths, they

\* need to verify that they were assigned a discrete or triggered

\* sample time. To do this:

\* mdlSetWorkWidths:

\* if (ssGetSampleTime(S, 0) == CONTINUOUS\_SAMPLE\_TIME) {

\* ssSetErrorStatus(S, "This block cannot be assigned a "

\* "continuous sample time");

\* }

\*

\* If your function has no intrinsic sample time, then you should indicate

\* that your sample time is inherited according to the following guidelines:

\*

\* o A function that changes as its input changes, even during minor

\* integration steps should register the [INHERITED\_SAMPLE\_TIME, 0.0]

\* sample time.

\* o A function that changes as its input changes, but doesn't change

\* during minor integration steps (i.e., 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),

\* you should use the ssIsSampleHit or ssIsContinuousTask macros.

\* For example, if your first sample time is continuous, then you

\* used the following code-fragment to check for a sample hit. Note,

\* you would get incorrect results if you used ssIsSampleHit(S,0,tid).

\* if (ssIsContinuousTask(S,tid)) {

\* }

\* If say, 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) {

\* }

\*

\*/

static void mdlInitializeSampleTimes(SimStruct \*S)

{

/\* Register one pair for each sample time \*/

ssSetSampleTime(S, 0, CONTINUOUS\_SAMPLE\_TIME);

ssSetOffsetTime(S, 0, 0.0);

} /\* end mdlInitializeSampleTimes \*/

#define MDL\_SET\_INPUT\_PORT\_DATA\_TYPE /\* Change to #undef to remove function \*/

#if defined(MDL\_SET\_INPUT\_PORT\_DATA\_TYPE) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetInputPortDataType =========================================

\* Abstract:

\* This method is called with the candidate data type id for a dynamically

\* typed input port. If the proposed data type is acceptable, the method

\* should go ahead and set the actual port data type using

\* ssSetInputPortDataType. If the data type is unacceptable an error

\* should generated via ssSetErrorStatus. Note that any other dynamically

\* typed input or output ports whose data types are implicitly defined by

\* virtue of knowing the data type of the given port can also have their

\* data types set via calls to ssSetInputPortDataType or

\* ssSetOutputPortDataType.

\*

\* See matlabroot/simulink/include/simstruc\_types.h for built-in

\* type defines: SS\_DOUBLE, SS\_BOOLEAN, etc.

\*

\* See matlabroot/simulink/src/sfun\_dtype\_io.c for an example.

\*/

static void mdlSetInputPortDataType(SimStruct \*S, int portIndex,DTypeId dType)

{

} /\* mdlSetInputPortDataType \*/

#endif /\* MDL\_SET\_INPUT\_PORT\_DATA\_TYPE \*/

#define MDL\_SET\_OUTPUT\_PORT\_DATA\_TYPE /\* Change to #undef to remove function \*/

#if defined(MDL\_SET\_OUTPUT\_PORT\_DATA\_TYPE) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetOutputPortDataType ========================================

\* Abstract:

\* This method is called with the candidate data type id for a dynamically

\* typed output port. If the proposed data type is acceptable, the method

\* should go ahead and set the actual port data type using

\* ssSetOutputPortDataType. If the data type is unacceptable an error

\* should generated via ssSetErrorStatus. Note that any other dynamically

\* typed input or output ports whose data types are implicitly defined by

\* virtue of knowing the data type of the given port can also have their

\* data types set via calls to ssSetInputPortDataType or

\* ssSetOutputPortDataType.

\*

\* See matlabroot/simulink/src/sfun\_dtype\_io.c for an example.

\*/

static void mdlSetOutputPortDataType(SimStruct \*S,int portIndex,DTypeId dType)

{

} /\* mdlSetOutputPortDataType \*/

#endif /\* MDL\_SET\_OUTPUT\_PORT\_DATA\_TYPE \*/

#define MDL\_SET\_DEFAULT\_PORT\_DATA\_TYPES /\* Change to #undef to remove function\*/

#if defined(MDL\_SET\_DEFAULT\_PORT\_DATA\_TYPES) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetDefaultPortDataTypes =====================================

\* Abstract:

\* This method is called when there is not enough information in your

\* model to uniquely determine the input and output data types

\* for your block. When this occurs, Simulink's data type propagation

\* engine calls this method to ask you to set your S-function default

\* data type for any dynamically typed input and output ports.

\*

\* If you do not provide this method and you have dynamically typed

\* ports where Simulink does not have enough information to propagate

\* data types to your S-function, then Simulink will assign the

\* data type to the largest known port data type of your S-function.

\* If there are no known data types, then Simulink will set the

\* data type to double.

\*

\* See matlabroot/simulink/src/sfun\_dtype\_io.c for an example.

\*/

static void mdlSetDefaultPortDataTypes(SimStruct \*S)

{

} /\* mdlSetDefaultPortDataTypes \*/

#endif /\* MDL\_SET\_DEFAULT\_PORT\_DATA\_TYPES \*/

#define MDL\_SET\_INPUT\_PORT\_COMPLEX\_SIGNAL /\* Change to #undef to remove \*/

#if defined(MDL\_SET\_INPUT\_PORT\_COMPLEX\_SIGNAL) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetInputPortComplexSignal ====================================

\* Abstract:

\* This method is called with the candidate complexity signal setting

\* (COMPLEX\_YES or COMPLEX\_NO) for an input port whos complex signal

\* attribute is set to COMPLEX\_INHERITED. If the proposed complexity is

\* acceptable, the method should go ahead and set the actual complexity

\* using ssSetInputPortComplexSignal. If the complex setting is

\* unacceptable an error should generated via ssSetErrorStatus. Note that

\* any other unknown ports whose complexity is implicitly defined by virtue

\* of knowing the complexity of the given port can also have their

\* complexity set via calls to ssSetInputPortComplexSignal or

\* ssSetOutputPortComplexSignal.

\*

\* See matlabroot/simulink/src/sfun\_cplx.c for an example.

\*/

static void mdlSetInputPortComplexSignal(SimStruct \*S,

int portIndex,

CSignal\_T cSignalSetting)

{

} /\* mdlSetInputPortComplexSignal \*/

#endif /\* MDL\_SET\_INPUT\_PORT\_COMPLEX\_SIGNAL \*/

#define MDL\_SET\_OUTPUT\_PORT\_COMPLEX\_SIGNAL /\* Change to #undef to remove \*/

#if defined(MDL\_SET\_OUTPUT\_PORT\_COMPLEX\_SIGNAL) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetOutputPortComplexSignal ===================================

\* Abstract:

\* This method is called with the candidate complexity signal setting

\* (COMPLEX\_YES or COMPLEX\_NO) for an output port whos complex signal

\* attribute is set to COMPLEX\_INHERITED. If the proposed complexity is

\* acceptable, the method should go ahead and set the actual complexity

\* using ssSetOutputPortComplexSignal. If the complex setting is

\* unacceptable an error should generated via ssSetErrorStatus. Note that

\* any other unknown ports whose complexity is implicitly defined by virtue

\* of knowing the complexity of the given port can also have their

\* complexity set via calls to ssSetInputPortComplexSignal or

\* ssSetOutputPortComplexSignal.

\*

\* See matlabroot/simulink/src/sfun\_cplx.c for an example.

\*/

static void mdlSetOutputPortComplexSignal(SimStruct \*S,

int portIndex,

CSignal\_T cSignalSetting)

{

} /\* mdlSetOutputPortComplexSignal \*/

#endif /\* MDL\_SET\_OUTPUT\_PORT\_COMPLEX\_SIGNAL \*/

#define MDL\_SET\_DEFAULT\_PORT\_COMPLEX\_SIGNALS /\* Change to #undef to remove \*/

#if defined(MDL\_SET\_DEFAULT\_PORT\_COMPLEX\_SIGNALS) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetDefaultPortComplexSignals ================================

\* Abstract:

\* This method is called when there is not enough information in your

\* model to uniquely determine the complexity (COMPLEX\_NO, COMPLEX\_YES)

\* of signals entering your block. When this occurs, Simulink's

\* complex signal propagation engine calls this method to ask you to set

\* your S-function default complexity type for any input and output ports

\* who's complex signal attribute is set to COMPLEX\_INHERITED.

\*

\* If you do not provide this method and you have COMPLEX\_INHERITED

\* ports where Simulink does not have enough information to propagate

\* the complexity to your S-function, then Simulink will set

\* these unkown ports to COMPLEX\_YES if any of your S-function

\* ports are currently set to COMPLEX\_YES, otherwise the unknown

\* ports will be set to COMPLEX\_NO.

\*

\* See matlabroot/simulink/src/sfun\_cplx.c for an example.

\*/

static void mdlSetDefaultPortComplexSignals(SimStruct \*S)

{

} /\* mdlSetDefaultPortComplexSignals \*/

#endif /\* MDL\_SET\_DEFAULT\_PORT\_COMPLEX\_SIGNALS \*/

#define MDL\_SET\_WORK\_WIDTHS /\* Change to #undef to remove function \*/

#if defined(MDL\_SET\_WORK\_WIDTHS) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlSetWorkWidths ===============================================

\* Abstract:

\* The optional method, 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 which are

\* a function of the input, output, and/or sample times. This method

\* is used to specify the nonzero work vector widths via the macros

\* ssNumContStates, ssSetNumDiscStates, ssSetNumRWork, ssSetNumIWork,

\* ssSetNumPWork, ssSetNumModes, and ssSetNumNonsampledZCs.

\*

\* Run-time parameters are registered in this method using methods

\* ssSetNumRunTimeParams, ssSetRunTimeParamInfo, and related methods.

\*

\* If you are using mdlSetWorkWidths, then any work vectors you are

\* using in your S-function should be set to DYNAMICALLY\_SIZED in

\* mdlInitializeSizes, even if the exact value is known at that point.

\* The actual size to be used by the S-function should then be specified

\* in mdlSetWorkWidths.

\*/

static void mdlSetWorkWidths(SimStruct \*S)

{

}

#endif /\* MDL\_SET\_WORK\_WIDTHS \*/

#define MDL\_INITIALIZE\_CONDITIONS /\* Change to #undef to remove function \*/

#if defined(MDL\_INITIALIZE\_CONDITIONS)

/\* Function: mdlInitializeConditions ========================================

\* Abstract:

\* In this function, you should initialize the continuous and discrete

\* states for your S-function block. The initial states are placed

\* in the state vector, ssGetContStates(S) or ssGetDiscStates(S).

\* You can also perform any other initialization activities that your

\* S-function may require. Note, this method will be called at the

\* start of simulation and if it is present in an enabled subsystem

\* configured to reset states, it will be call when the enabled subsystem

\* restarts execution to reset the states.

\*

\* You can use the ssIsFirstInitCond(S) macro to determine if this is

\* is the first time mdlInitializeConditions is being called.

\*/

static void mdlInitializeConditions(SimStruct \*S)

{

}

#endif /\* MDL\_INITIALIZE\_CONDITIONS \*/

#define MDL\_START /\* Change to #undef to remove function \*/

#if defined(MDL\_START)

/\* Function: mdlStart =======================================================

\* Abstract:

\* This function is called once at start of model execution. If you

\* have states that should be initialized once, this is the place

\* to do it.

\*/

static void mdlStart(SimStruct \*S)

{

}

#endif /\* MDL\_START \*/

/\* Define to indicate that this S-Function has the mdlG[S]etSimState mothods \*/

#define MDL\_SIM\_STATE /\* Change to #undef to remove this function \*/

#if defined(MDL\_SIM\_STATE)

/\* Function: mdlGetSimState ===========================================

\* Abstract:

\* Package complete simulation state (this includes all the run-time data

\* that can change \*after\* mdlStart) as a MATLAB data structure and return

\* it.

\*/

static mxArray\* mdlGetSimState(SimStruct\* S)

{

}

/\* Function: mdlSetSimState ===========================================

\* Abstract:

\* Unack inSimState MATLAB data structure which contains the complete

\* simulation state (this includes all the run-time data that can change

\* \*after\* mdlStart) into the appropriate locations.

\*/

static void mdlSetSimState(SimStruct\* S, const mxArray\* inSimState)

{

}

#endif /\* MDL\_SIM\_STATE \*/

#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))

/\* Function: mdlGetTimeOfNextVarHit =========================================

\* Abstract:

\* This function 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 time of next hit by using ssSetTNext. The time of

\* the next hit must be greater than ssGetT(S).

\*

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

\*/

static void mdlGetTimeOfNextVarHit(SimStruct \*S)

{

time\_T timeOfNextHit = ssGetT(S) /\* + offset \*/ ;

ssSetTNext(S, timeOfNextHit);

}

#endif /\* MDL\_GET\_TIME\_OF\_NEXT\_VAR\_HIT \*/

#define MDL\_ZERO\_CROSSINGS /\* Change to #undef to remove function \*/

#if defined(MDL\_ZERO\_CROSSINGS) && (defined(MATLAB\_MEX\_FILE) || defined(NRT))

/\* Function: mdlZeroCrossings ===============================================

\* Abstract:

\* If your S-function has registered CONTINUOUS\_SAMPLE\_TIME and there

\* are signals entering the S-function or internally generated signals

\* which have discontinuities, you can use this method to locate the

\* discontinuities. When called, this method must update the

\* ssGetNonsampleZCs(S) vector.

\*/

static void mdlZeroCrossings(SimStruct \*S)

{

}

#endif /\* MDL\_ZERO\_CROSSINGS \*/

/\* Function: mdlOutputs =======================================================

\* Abstract:

\* In this function, you compute the outputs of your S-function

\* block. Generally outputs are placed in the output vector(s),

\* ssGetOutputPortSignal.

\*/

static void mdlOutputs(SimStruct \*S, int\_T tid)

{

} /\* end mdlOutputs \*/

#define MDL\_UPDATE /\* Change to #undef to remove function \*/

#if defined(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)

{

}

#endif /\* MDL\_UPDATE \*/

#define MDL\_DERIVATIVES /\* Change to #undef to remove function \*/

#if defined(MDL\_DERIVATIVES)

/\* Function: mdlDerivatives =================================================

\* Abstract:

\* In this function, you compute the S-function block's derivatives.

\* The derivatives are placed in the derivative vector, ssGetdX(S).

\*/

static void mdlDerivatives(SimStruct \*S)

{

}

#endif /\* MDL\_DERIVATIVES \*/

/\* Function: mdlTerminate =====================================================

\* Abstract:

\* In this function, you should perform any actions that are necessary

\* at the termination of a simulation. For example, if memory was allocated

\* in mdlStart, this is the place to free it.

\*

\* Suppose your S-function allocates a few few chunks of memory in mdlStart

\* and saves them in PWork. The following code fragment would free this

\* memory.

\* {

\* int i;

\* for (i = 0; i<ssGetNumPWork(S); i++) {

\* if (ssGetPWorkValue(S,i) != NULL) {

\* free(ssGetPWorkValue(S,i));

\* }

\* }

\* }

\*/

static void mdlTerminate(SimStruct \*S)

{

}

#define MDL\_RTW /\* Change to #undef to remove function \*/

#if defined(MDL\_RTW) && defined(MATLAB\_MEX\_FILE)

/\* Function: mdlRTW =========================================================

\* Abstract:

\*

\* This function is called when the Real-Time Workshop is generating

\* the model.rtw file. In this method, you can call the following

\* functions which add fields to the model.rtw file.

\*

\* 1) The following creates Parameter records for your S-functions.

\* nParams is the number of tunable S-function parameters.

\*

\* if ( !ssWriteRTWParameters(S, nParams,

\*

\* SSWRITE\_VALUE\_[type],paramName,stringInfo,

\* [type specific arguments below]

\*

\* ) ) {

\* return; (error reporting will be handled by SL)

\* }

\*

\* Where SSWRITE\_VALUE\_[type] can be one of the following groupings

\* (and you must have "nParams" such groupings):

\*

\* SSWRITE\_VALUE\_VECT,

\* const char\_T \*paramName,

\* const char\_T \*stringInfo,

\* const real\_T \*valueVect,

\* int\_T vectLen

\*

\* SSWRITE\_VALUE\_2DMAT,

\* const char\_T \*paramName,

\* const char\_T \*stringInfo,

\* const real\_T \*valueMat,

\* int\_T nRows,

\* int\_T nCols

\*

\* SSWRITE\_VALUE\_DTYPE\_VECT,

\* const char\_T \*paramName,

\* const char\_T \*stringInfo,

\* const void \*valueVect,

\* int\_T vectLen,

\* int\_T dtInfo

\*

\* SSWRITE\_VALUE\_DTYPE\_2DMAT,

\* const char\_T \*paramName,

\* const char\_T \*stringInfo,

\* const void \*valueMat,

\* int\_T nRows,

\* int\_T nCols,

\* int\_T dtInfo

\*

\* SSWRITE\_VALUE\_DTYPE\_ML\_VECT,

\* const char\_T \*paramName,

\* const char\_T \*stringInfo,

\* const void \*rValueVect,

\* const void \*iValueVect,

\* int\_T vectLen,

\* int\_T dtInfo

\*

\* SSWRITE\_VALUE\_DTYPE\_ML\_2DMAT,

\* const char\_T \*paramName,

\* const char\_T \*stringInfo,

\* const void \*rValueMat,

\* const void \*iValueMat,

\* int\_T nRows,

\* int\_T nCols,

\* int\_T dtInfo

\*

\* Notes:

\* 1. nParams is an integer and stringInfo is a string describing

\* generalinformation about the parameter such as how it was derived.

\* 2. The last argument to this function, dtInfo, is obtained from the

\* DTINFO macro (defined in simstruc.h) as:

\* dtInfo = DTINFO(dataTypeId, isComplexSignal);

\* where dataTypeId is the data type id and isComplexSignal is a

\* boolean value specifying whether the parameter is complex.

\*

\* See simulink/include/simulink.c for the definition (implementation)

\* of this function and simulink/src/sfun\_multiport.c for an example

\* of using this function.

\*

\* 2) The following creates SFcnParameterSetting record for S-functions

\* (these can be derived from the non-tunable S-function parameters).

\*

\* if ( !ssWriteRTWParamSettings(S, nParamSettings,

\*

\* SSWRITE\_VALUE\_[whatever], settingName,

\* [type specific arguments below]

\*

\* ) ) {

\* return; (error reporting will be handled by SL)

\* }

\*

\* Where SSWRITE\_VALUE\_[type] can be one of the following groupings

\* (and you must have "nParamSettings" such groupings):

\* Also, the examples in the right hand column below show how the

\* ParamSetting appears in the .rtw file

\*

\* SSWRITE\_VALUE\_STR, - Used to write (un)quoted strings

\* const char\_T \*settingName, example:

\* const char\_T \*value, Country USA

\*

\* SSWRITE\_VALUE\_QSTR, - Used to write quoted strings

\* const char\_T \*settingName, example:

\* const char\_T \*value, Country "U.S.A"

\*

\* SSWRITE\_VALUE\_VECT\_STR, - Used to write vector of strings

\* const char\_T \*settingName, example:

\* const char\_T \*value, Countries ["USA", "Mexico"]

\* int\_T nItemsInVect

\*

\* SSWRITE\_VALUE\_NUM, - Used to write numbers

\* const char\_T \*settingName, example:

\* const real\_T value NumCountries 2

\*

\*

\* SSWRITE\_VALUE\_VECT, - Used to write numeric vectors

\* const char\_T \*settingName, example:

\* const real\_T \*settingValue, PopInMil [300, 100]

\* int\_T vectLen

\*

\* SSWRITE\_VALUE\_2DMAT, - Used to write 2D matrices

\* const char\_T \*settingName, example:

\* const real\_T \*settingValue, PopInMilBySex Matrix(2,2)

\* int\_T nRows, [[170, 130],[60, 40]]

\* int\_T nCols

\*

\* SSWRITE\_VALUE\_DTYPE\_NUM, - Used to write numeric vectors

\* const char\_T \*settingName, example: int8 Num 3+4i

\* const void \*settingValue, written as: [3+4i]

\* int\_T dtInfo

\*

\*

\* SSWRITE\_VALUE\_DTYPE\_VECT, - Used to write data typed vectors

\* const char\_T \*settingName, example: int8 CArray [1+2i 3+4i]

\* const void \*settingValue, written as:

\* int\_T vectLen CArray [1+2i, 3+4i]

\* int\_T dtInfo

\*

\*

\* SSWRITE\_VALUE\_DTYPE\_2DMAT, - Used to write data typed 2D

\* const char\_T \*settingName matrices

\* const void \*settingValue, example:

\* int\_T nRow , int8 CMatrix [1+2i 3+4i; 5 6]

\* int\_T nCols, written as:

\* int\_T dtInfo CMatrix Matrix(2,2)

\* [[1+2i, 3+4i]; [5+0i, 6+0i]]

\*

\*

\* SSWRITE\_VALUE\_DTYPE\_ML\_VECT, - Used to write complex matlab data

\* const char\_T \*settingName, typed vectors example:

\* const void \*settingRValue, example: int8 CArray [1+2i 3+4i]

\* const void \*settingIValue, settingRValue: [1 3]

\* int\_T vectLen settingIValue: [2 4]

\* int\_T dtInfo

\* written as:

\* CArray [1+2i, 3+4i]

\*

\* SSWRITE\_VALUE\_DTYPE\_ML\_2DMAT, - Used to write matlab complex

\* const char\_T \*settingName, data typed 2D matrices

\* const void \*settingRValue, example

\* const void \*settingIValue, int8 CMatrix [1+2i 3+4i; 5 6]

\* int\_T nRows settingRValue: [1 5 3 6]

\* int\_T nCols, settingIValue: [2 0 4 0]

\* int\_T dtInfo

\* written as:

\* CMatrix Matrix(2,2)

\* [[1+2i, 3+4i]; [5+0i, 6+0i]]

\*

\* Note, The examples above show how the ParamSetting is written out

\* to the .rtw file

\*

\* See simulink/include/simulink.c for the definition (implementation)

\* of this function and simulink/src/sfun\_multiport.c for an example

\* of using this function.

\*

\* 3) The following creates the work vector records for S-functions

\*

\* if (!ssWriteRTWWorkVect(S, vectName, nNames,

\*

\* name, size, (must have nNames of these pairs)

\* :

\* ) ) {

\* return; (error reporting will be handled by SL)

\* }

\*

\* Notes:

\* a) vectName must be either "RWork", "IWork" or "PWork"

\* b) nNames is an int\_T (integer), name is a const char\_T\* (const

\* char pointer) and size is int\_T, and there must be nNames number

\* of [name, size] pairs passed to the function.

\* b) intSize1+intSize2+ ... +intSizeN = ssGetNum<vectName>(S)

\* Recall that you would have to set ssSetNum<vectName>(S)

\* in one of the initialization functions (mdlInitializeSizes

\* or mdlSetWorkVectorWidths).

\*

\* See simulink/include/simulink.c for the definition (implementation)

\* of this function, and ... no example yet :(

\*

\* 4) Finally the following functions/macros give you the ability to write

\* arbitrary strings and [name, value] pairs directly into the .rtw

\* file.

\*

\* if (!ssWriteRTWStr(S, const\_char\_\*\_string)) {

\* return;

\* }

\*

\* if (!ssWriteRTWStrParam(S, const\_char\_\*\_name, const\_char\_\*\_value)) {

\* return;

\* }

\*

\* if (!ssWriteRTWScalarParam(S, const\_char\_\*\_name,

\* const\_void\_\*\_value,

\* DTypeId\_dtypeId)) {

\* return;

\* }

\*

\* if (!ssWriteRTWStrVectParam(S, const\_char\_\*\_name,

\* const\_char\_\*\_value,

\* int\_num\_items)) {

\* return;

\* }

\*

\* if (!ssWriteRTWVectParam(S, const\_char\_\*\_name, const\_void\_\*\_value,

\* int\_data\_type\_of\_value, int\_vect\_len)){

\* return;

\* }

\*

\* if (!ssWriteRTW2dMatParam(S, const\_char\_\*\_name, const\_void\_\*\_value,

\* int\_data\_type\_of\_value, int\_nrows, int\_ncols)){

\* return;

\* }

\*

\* The 'data\_type\_of\_value' input argument for the above two macros is

\* obtained using

\* DTINFO(dTypeId, isComplex),

\* where

\* dTypeId: can be any one of the enum values in BuitlInDTypeID

\* (SS\_DOUBLE, SS\_SINGLE, SS\_INT8, SS\_UINT8, SS\_INT16,

\* SS\_UINT16, SS\_INT32, SS\_UINT32, SS\_BOOLEAN defined

\* in simstuc\_types.h)

\* isComplex: is either 0 or 1, as explained in Note-2 for

\* ssWriteRTWParameters.

\*

\* For example DTINFO(SS\_INT32,0) is a non-complex 32-bit signed

\* integer.

\*

\* If isComplex==1, then it is assumed that 'const\_void\_\*\_value' array

\* has the real and imaginary parts arranged in an interleaved manner

\* (i.e., Simulink Format).

\*

\* If you prefer to pass the real and imaginary parts as two seperate

\* arrays, you should use the follwing macros:

\*

\* if (!ssWriteRTWMxVectParam(S, const\_char\_\*\_name,

\* const\_void\_\*\_rvalue, const\_void\_\*\_ivalue,

\* int\_data\_type\_of\_value, int\_vect\_len)){

\* return;

\* }

\*

\* if (!ssWriteRTWMx2dMatParam(S, const\_char\_\*\_name,

\* const\_void\_\*\_rvalue, const\_void\_\*\_ivalue,

\* int\_data\_type\_of\_value,

\* int\_nrows, int\_ncols)){

\* return;

\* }

\*

\* See simulink/include/simulink.c and simstruc.h for the definition

\* (implementation) of these functions and simulink/src/ml2rtw.c for

\* examples of using these functions.

\*

\*/

static void mdlRTW(SimStruct \*S)

{

}

#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