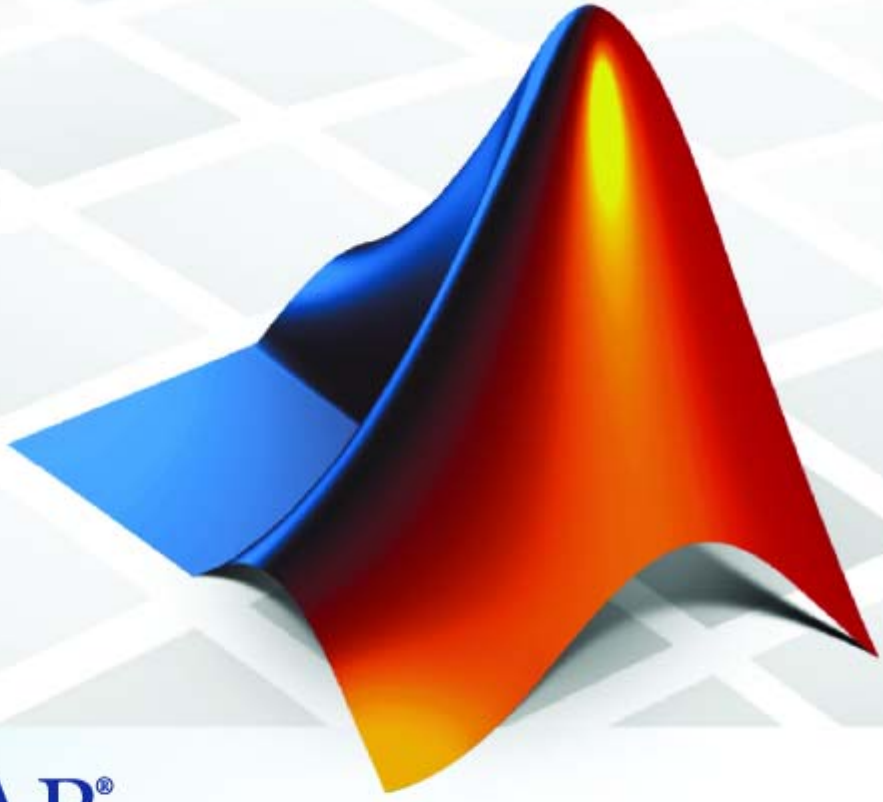
Real-Time Workshop® Embedded Coder 4
Reference



MATLAB®
&SIMULINK®

The MathWorks
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Real-Time Workshop Embedded Coder Reference

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

September 2006	Online only	New for Version 4.5 (Release 2006b)
March 2007	Online only	Revised for Version 4.6 (Release 2007a)

Contents

Index

1

Functions — By Category

Function Prototype Control

`addArgConf`	Add argument configuration information for Simulink model port to model-specific C function prototype
`attachToModel`	Attach model-specific C function prototype to loaded ERT-based Simulink model
`getArgCategory`	Get argument category for Simulink model port from model-specific C function prototype
`getArgName`	Get argument name for Simulink model port from model-specific C function prototype
`getArgPosition`	Get argument position for Simulink model port from model-specific C function prototype
`getArgQualifier`	Get argument type qualifier for Simulink model port from model-specific C function prototype
`getDefaultConf`	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
`getFunctionName`	Get function name from model-specific C function prototype
`getNumArgs`	Get number of function arguments from model-specific C function prototype
`runValidation`	Validate model-specific C function prototype against Simulink model to which it is attached

`setArgCategory`	Set argument category for Simulink model port in model-specific C function prototype
`setArgName`	Set argument name for Simulink model port in model-specific C function prototype
`setArgPosition`	Set argument position for Simulink model port in model-specific C function prototype
`setArgQualifier`	Set argument type qualifier for Simulink model port in model-specific C function prototype
`setFunctionName`	Set function name in model-specific C function prototype

Model Entry Points

`model_initialize`	Initialization entry point in generated code for ERT-based Simulink model
`model_SetEventsForThisBaseStep`	Set event flags for multirate, multitasking operation before calling *`model`*`_step` for ERT-based Simulink model
`model_step`	Step routine entry point in generated code for ERT-based Simulink model
`model_terminate`	Termination entry point in generated code for ERT-based Simulink model

System Target File Callback Interface

`slConfigUIGetVal`	Return current value for custom target configuration option
`slConfigUISetEnabled`	Enable or disable custom target configuration option
`slConfigUISetVal`	Set value for custom target configuration option

2

Functions — Alphabetical List

addArgConf

Purpose

Add argument configuration information for Simulink model port to model-specific C function prototype

Syntax

```
addArgConf(obj, portName, category, argName, qualifier)
```

Arguments

`obj`
: Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

`portName`
: String specifying the unqualified name of an inport or outport in your Simulink model.

`category`
: String specifying the argument category, either `'Value'` or `'Pointer'`.

`argName`
: String specifying a valid C identifier.

`qualifier`
: String specifying the argument type qualifier: `'none'`, `'const'`, `'const *'`, or `'const * const'`.

Description

The `addArgConf` function adds argument configuration information for a port in your ERT-based Simulink model to a model-specific C function prototype. You specify the name of the model port, the argument category (`'Value'` or `'Pointer'`), the argument name, and the argument type qualifier (for example, `'const'`).

The order of `addArgConf` calls will determine the argument position for the port in the function prototype, unless it is changed by other means.

If a port has an existing argument configuration, subsequent calls to `addArgConf` with the same port name will overwrite the port's previous argument configuration.

Example

In the following example, the `addArgConf` function is used to add argument configuration information for ports `Input` and `Output` in an ERT-based version of `rtwdemo_counter`. After executing these commands, you can click the **Configure Functions** button on the **Interface** pane of the Configuration Parameters dialog box to bring up the Model Step Function dialog box and confirm that the `addArgConf` commands succeeded.

```
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a function control object
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the function control object to the model
attachToModel(a,gcs)
```

See Also

`attachToModel`

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

attachToModel

Purpose

Attach model-specific C function prototype to loaded ERT-based Simulink model

Syntax

```
attachToModel(obj, modelName)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.ModelSpecificCPrototype`.

`modelName`

String specifying the name of a loaded ERT-based Simulink model to which the object is going to be attached.

Description

The `attachToModel` function attaches a model-specific C function prototype to a loaded ERT-based Simulink model.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get argument category for Simulink model port from model-specific C function prototype

Syntax

```
category = getArgCategory(obj, portName)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

`portName`

String specifying the name of an inport or outport in your Simulink model.

Returns

String specifying the argument category, `'Value'` or `'Pointer'`, for the specified Simulink model port.

Description

The `getArgCategory` function gets the category (`'Value'` or `'Pointer'`) of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getArgName

Purpose

Get argument name for Simulink model port from model-specific C function prototype

Syntax

```
argName = getArgName(obj, portName)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

`portName`

String specifying the name of an inport or outport in your Simulink model.

Returns

String specifying the argument name for the specified Simulink model port.

Description

The `getArgName` function gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get argument position for Simulink model port from model-specific C function prototype

Syntax

```
position = getArgPosition(obj, portName)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

`portName`

String specifying the name of an inport or outport in your Simulink model.

Returns

Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. If no argument is found for the specified port, the function returns 0.

Description

The `getArgPosition` function gets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getArgQualifier

Purpose

Get argument type qualifier for Simulink model port from model-specific C function prototype

Syntax

```
qualifier = getArgQualifier(obj, portName)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

`portName`

String specifying the name of an inport or outport in your Simulink model.

Returns

String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — for the specified Simulink model port.

Description

The `getArgQualifier` function gets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get default configuration information for model-specific C function prototype from Simulink model to which it is attached

Syntax

```
getDefaultConf(obj)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.ModelSpecificCPrototype`.

Description

The `getDefaultConf` function initializes the specified model-specific C function prototype to a default configuration based on information from the ERT-based Simulink model to which it is attached.

Before calling this function, you must call `attachToModel`, to attach the function prototype to a loaded model.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getFunctionName

Purpose

Get function name from model-specific C function prototype

Syntax

```
fcnName = getFunctionName(obj)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

Returns

A string specifying the name of the function described by the specified model-specific C function prototype.

Description

The `getFunctionName` function gets the name of the function described by the specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get number of function arguments from model-specific C function prototype

Syntax

```
num = getNumArgs(obj)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object, such as a handle previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

Returns

An integer specifying the number of function arguments.

Description

The `getNumArgs` function gets the number of function arguments for the function described by the specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

model_initialize

Purpose

Initialization entry point in generated code for ERT-based Simulink model

Syntax

```
void model_initialize(void)
void model_initialize(boolean_T firstTime)
```

Arguments

`firstTime`

Real-Time Workshop Embedded Coder generates this argument for Simulink models only if the `IncludeERTFirstTime` model configuration parameter is set to `on`. Use of the `firstTime` argument will be discontinued in a future release (see the note below).

Specifies value 0 (FALSE) or 1 (TRUE). If `firstTime` equals 1, *`model`*`_initialize` initializes `rtModel` and other data structures private to the model. If `firstTime` equals 0, *`model`*`_initialize` resets the model's states, but does not initialize other data structures. Call *`model`*`_initialize` with `firstTime` set to 0 to reset the model's states at a time greater than start time.

Description

The *`model`*`_initialize` function contains all model initialization code. The generated code for a Simulink model calls *`model`*`_initialize` once, at the beginning of model execution.

If the `IncludeERTFirstTime` model configuration parameter is set to `on`, the generated code passes in `firstTime` as `1` (`TRUE`).

Note In a future release, Real-Time Workshop Embedded Coder will no longer use the `firstTime` argument in a model's generated *`model`*`_initialize` function. For more information about the `IncludeERTFirstTime` model configuration parameter and a related target configuration parameter, `ERTFirstTimeCompliant`, see "Configuration Parameter Reference" in the Real-Time Workshop documentation.

See Also

`model_SetEventsForThisBaseStep`, `model_step`, `model_terminate`

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

model_SetEventsForThisBaseStep

Purpose

Set event flags for multirate, multitasking operation before calling *model*_step for ERT-based Simulink model

Syntax

```
void model_SetEventsForThisBaseStep(boolean_T *eventFlags)
void model_SetEventsForThisBaseStep(boolean_T *eventFlags,
RT_MODEL_model *model_M)
```

Arguments

`eventFlags`
: Pointer to the model's event flags array.

model`_M`
: Pointer to the real-time model object. Real-Time Workshop Embedded Coder generates this argument only if **Generate reusable code** is on.

Description

Real-Time Workshop Embedded Coder generates the *model*`_SetEventsForThisBaseStep` utility function only for multirate, multitasking models.

The *model*`_SetEventsForThisBaseStep` function maintains model event flags that determine which subrate tasks need to run on a given base rate time step. In a multirate, multitasking application, the program code must call *model*`_SetEventsForThisBaseStep` before calling the *model*`_step` function. See "Multirate Multitasking Operation" in the Real-Time Workshop Embedded Coder documentation for further information.

Note The macro `MODEL_SETEVENTS`, defined in the static `ert_main.c` module, provides a way to call *model*`_SetEventsForThisBaseStep` from a static main program.

See Also

`model_initialize`, `model_step`, `model_terminate`

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

Purpose

Step routine entry point in generated code for ERT-based Simulink model

Syntax

```
void model_step(void)
void model_step(int_T tid)
void model_stepN(void)
```

Arguments

`tid`

Task identifier. Real-Time Workshop Embedded Coder generates this argument only for multirate, single-tasking models.

Calling Interfaces

The *model*_step default function prototype varies depending on the number of rates in the model and the solver mode, as shown below:

Rates/Solver Mode	Function Prototype
Single-rate/SingleTasking	`void model_step(void);`
Multirate/SingleTasking	`void model_step(int_T tid);`
Multirate/MultiTasking (rate grouping)	`void model_stepN (void);` (*N* is a task identifier)

If you generate reusable, reentrant code for an ERT-based model using the **Generate reusable code** option, the generated code passes the model's root-level inputs and outputs, block states, parameters, and external outputs to *model*_step using a function prototype that generally resembles the following:

```
void model_step(inport_args, outport_args, BlockIO_arg,
DWork_arg, RT_model_arg);
```

The manner in which the inport and outport arguments are passed is determined by the setting of the **Pass root-level I/O as** parameter, which appears on the **Interface** pane of the Configuration Parameters dialog box only if **Generate reusable code** is selected.

For greater control over the *model*_step function prototype, you can use the **Configure Functions** button on the **Interface** pane to launch a

Model Step Functions dialog box (see "Model Step Functions Dialog Box" in the Real-Time Workshop Embedded Coder documentation). Based on the **Function specification** value you specify for your *model*_step function (supported values include `Default model-step function` and `Model specific C prototype`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications. For more information about controlling the *model*_step function prototype, see the sections "Interface Pane" and "Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation.

Description

Real-Time Workshop Embedded Coder generates the *model*_step function for a Simulink model when the **Single output/update function** configuration option is selected (the default) in the Configuration Parameters dialog box. *model*_step contains the output and update code for all blocks in the model.

*model*_step is designed to be called at interrupt level from rt_OneStep, which is assumed to be invoked as a timer ISR. rt_OneStep calls *model*_step to execute processing for one clock period of the model. See "rt_OneStep" in the Real-Time Workshop Embedded Coder documentation for a description of how calls to *model*_step are generated and scheduled.

Note If the **Single output/update function** configuration option is not selected, Real-Time Workshop Embedded Coder generates the following model entry point functions in place of *model*_step:

- *model*_output: Contains the output code for all blocks in the model
- *model*_update: Contain the update code for all blocks in the model

---

The *model*_step function computes the current value of all blocks. If logging is enabled, *model*_step updates logging variables. If the model's

stop time is finite, *model*_step signals the end of execution when the current time equals the stop time.

In cases where a tid is passed in, the caller (rt_OneStep) assigns each task a tid, and *model*_step uses the tid argument to determine which blocks have a sample hit (and, therefore, should execute).

Under any of the following conditions, *model*_step does not check the current time against the stop time:

- The model's stop time is set to inf.
- Logging is disabled.
- The **Terminate function required** option is not selected.

Therefore, if any of these conditions are true, the program runs indefinitely.

See Also

model_initialize, model_SetEventsForThisBaseStep, model_terminate

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

model_terminate

Purpose

Termination entry point in generated code for ERT-based Simulink model

Syntax

```
void model_terminate(void)
```

Description

Real-Time Workshop Embedded Coder generates the *model*_terminate function for a Simulink model when the **Terminate function required** configuration option is selected (the default) in the Configuration Parameters dialog box. *model*_terminate contains all model termination code and should be called as part of system shutdown.

When *model*_terminate is called, blocks that have a terminate function execute their terminate code. If logging is enabled, *model*_terminate ends data logging.

The *model*_terminate function should be called only once.

If your application runs indefinitely, you do not need the *model*_terminate function. To suppress the function, clear the **Terminate function required** configuration option in the Configuration Parameters dialog box.

See Also

model_initialize, model_SetEventsForThisBaseStep, model_step

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

Purpose

Validate model-specific C function prototype against Simulink model to which it is attached

Syntax

```
[status, msg] = runValidation(obj)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

Returns

`[status, msg]`, where `status` is true for a valid configuration and false otherwise. If `status` is false, `message` contains information explaining why the configuration is invalid.

Description

The `runValidation` function runs a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getFunctionSpecification`, to get the handle to a function prototype previously attached to a loaded model.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

setArgCategory

Purpose

Set argument category for Simulink model port in model-specific C function prototype

Syntax

```
setArgCategory(obj, portName, category)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

`portName`

String specifying the unqualified name of an inport or outport in your Simulink model.

`category`

String specifying the argument category, `'Value'` or `'Pointer'`, to be set for the specified Simulink model port.

Note If you change the argument category for an outport from `'Pointer'` to `'Value'`, the change will cause the argument to move to the first argument position when `attachToModel` or `runValidation` is called.

Description

The `setArgCategory` function sets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument name for Simulink model port in model-specific C function prototype

Syntax

```
setArgName(obj, portName, argName)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

`portName`

String specifying the name of an inport or outport in your Simulink model.

`argName`

String specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Description

The `setArgName` function sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

setArgPosition

Purpose

Set argument position for Simulink model port in model-specific C function prototype

Syntax

```
setArgPosition(obj, portName, position)
```

Arguments

`obj`
: Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

`portName`
: String specifying the name of an inport or outport in your Simulink model.

`position`
: Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Description

The `setArgPosition` function sets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype. The specified argument will be moved to the specified position, and other arguments will be shifted by one position accordingly.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument type qualifier for Simulink model port in model-specific C function prototype

Syntax

```
setArgQualifier(obj, portName, qualifier)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

`portName`

String specifying the name of an inport or outport in your Simulink model.

`qualifier`

String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — to be set for the specified Simulink model port.

Description

The `setArgQualifier` function sets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

setFunctionName

Purpose

Set function name in model-specific C function prototype

Syntax

```
setFunctionName(obj, fcnName)
```

Arguments

`obj`

Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

`fcnName`

String specifying a new name for the function described by the function control object. The argument must be a valid C identifier.

Description

The `setFunctionName` function sets the function name in the specified function control object.

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Return current value for custom target configuration option

Syntax

```
value = slConfigUIGetVal(hDlg, hSrc, 'OptionName')
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

Returns

Current value of the specified option. The data type of the return value depends on the data type of the option.

Description

The `slConfigUIGetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUIGetVal` to read the current value of a specified target option.

Example

In the following example, the `slConfigUIGetVal` function returns the value of the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
```

```
        slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

    slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
    slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUISetEnabled`, `slConfigUISetVal`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop Embedded Coder documentation

"Configuration Parameter Reference" in the Real-Time Workshop documentation

Purpose

Enable or disable custom target configuration option

Syntax

```
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', true)
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', false)
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`true`
: Specifies that the option should be enabled.

`false`
: Specifies that the option should be disabled.

Description

The `slConfigUISetEnabled` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetEnabled` to enable or disable a specified target option.

Example

In the following example, the `slConfigUISetEnabled` function disables the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
```

```
        '  Uncheck and disable "Terminate function required".']);

disp(['Value of IncludeMdlTerminateFcn was ', ...
      slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUIGetVal`, `slConfigUISetVal`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop Embedded Coder documentation

"Configuration Parameter Reference" in the Real-Time Workshop documentation

Purpose

Set value for custom target configuration option

Syntax

```
slConfigUISetVal(hDlg, hSrc, 'OptionName', OptionValue)
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`OptionValue`
: Value to be set for the specified option.

Description

The `slConfigUISetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetVal` to set the value of a specified target option.

Example

In the following example, the `slConfigUISetVal` function sets the value `'off'` for the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
        slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);
```

```
slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUIGetVal`, `slConfigUISetEnabled`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop Embedded Coder documentation

"Configuration Parameter Reference" in the Real-Time Workshop documentation

3

Blocks — By Category

Configuration Wizards

Custom M-file	Automatically update active configuration parameters of parent model using custom M-file
ERT (optimized for fixed-point)	Automatically update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Automatically update active configuration parameters of parent model for ERT floating-point code generation
GRT (debug for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

Module Packaging

Data Object Wizard	Simulink data object wizard for creating potential Simulink data objects

4

Blocks — Alphabetical List

Custom M-file

Purpose

Automatically update active configuration parameters of parent model using custom M-file

Library

Configuration Wizards

Description

Custom M-file

When you add a Custom M-file block to your Simulink model and double-click it, a custom M-file script executes and automatically configures model parameters that are relevant to code generation. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

The MathWorks provides an example M-file script, *matlabroot*`/toolbox/rtw/rtw/rtwsampleconfig.m`, that you can use with the Custom M-file block and adapt to your model requirements. The block and the script provide a starting point for customization. For more information, see "Creating a Custom Configuration Wizard Block" in the Real-Time Workshop Embedded Coder documentation.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `Custom` is selected by default.

Configuration function

Name of the predefined or custom M-file script to be used to update the active configuration parameters of the parent Simulink model. The default value is `rtwsampleconfig`, which refers to the example M-file script `rtwsampleconfig.m`.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

Data Object Wizard

Purpose

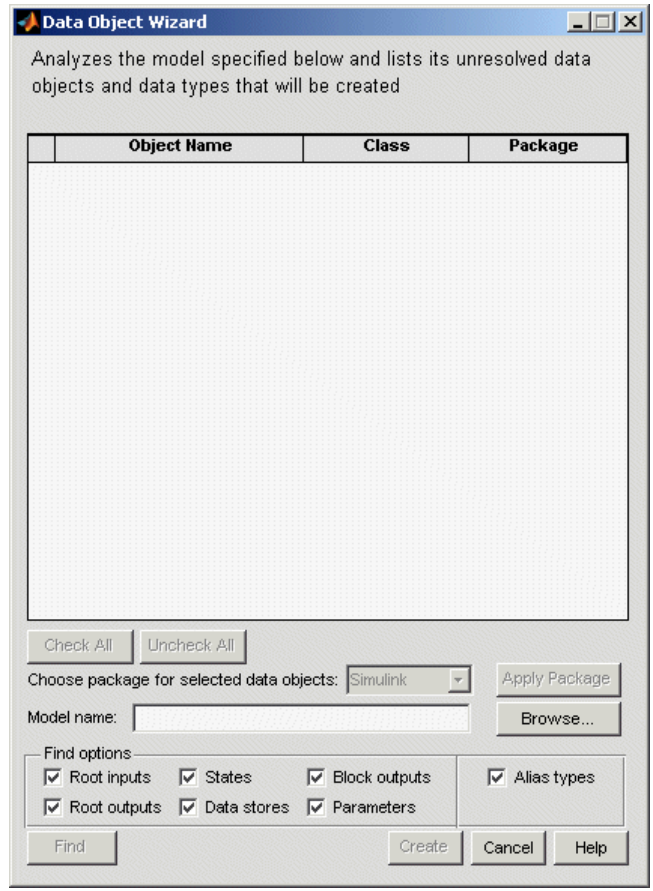
Simulink data object wizard for creating potential Simulink data objects

Library

Module Packaging

Description

When you add a Data Object Wizard block to your Simulink model and double-click it, the Data Object Wizard is launched:



The Data Object Wizard allows you to determine quickly which model data is not associated with Simulink data objects and to create and associate data objects with the data.

For detailed information about using the Data Object Wizard, see "Data Object Wizard" in the Simulink documentation and "Creating Simulink Data Objects with Data Object Wizard" in the Real-Time Workshop Embedded Coder documentation.

You can also launch the Data Object Wizard by entering `dataobjectwizard` at the MATLAB® command line or by selecting **Data Object Wizard** from the **Tools** menu of your model.

Example

For an example of a model that incorporates the Data Object Wizard block, see `rtwdemo_mpf`.

See Also

"Data Object Wizard" in the Simulink documentation

"Creating Simulink Data Objects with Data Object Wizard" in the Real-Time Workshop Embedded Coder documentation

"Creating a Data Dictionary for a Model" in the Real-Time Workshop Embedded Coder documentation

"Customizing Data Object Wizard User Packages" in the Real-Time Workshop Embedded Coder documentation

ERT (optimized for fixed-point)

Purpose

Automatically update active configuration parameters of parent model for ERT fixed-point code generation

Library

Configuration Wizards

Description

ERT (optimized for fixed-point)

When you add an ERT (optimized for fixed-point) block to your Simulink model and double-click it, a predefined M-file script executes and automatically configures the model parameters optimally for fixed-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `ERT (optimized for fixed-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for floating-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

ERT (optimized for floating-point)

Purpose

Automatically update active configuration parameters of parent model for ERT floating-point code generation

Library

Configuration Wizards

Description

When you add an ERT (optimized for floating-point) block to your Simulink model and double-click it, a predefined M-file script executes and automatically configures the model parameters optimally for floating-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `ERT (optimized for floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for fixed-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

GRT (debug for fixed/floating-point)

Purpose

Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled

Library

Configuration Wizards

Description

When you add a GRT (debug for fixed/floating-point) block to your Simulink model and double-click it, a predefined M-file script executes and automatically configures the model parameters optimally for fixed/floating-point code generation, with TLC debugging options enabled, with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `GRT (debug for fixed/floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

GRT (optimized for fixed/floating-point)

Purpose

Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

Library

Configuration Wizards

Description

When you add a GRT (optimized for fixed/floating-point) block to your Simulink model and double-click it, a predefined M-file script executes and automatically configures the model parameters optimally for fixed/floating-point code generation with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `GRT (optimized for fixed/floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (debug for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

5

Configuration Parameters Dialog Box Reference

Optimization Pane

Application lifespan (days)

Control the allocation of memory for absolute and elapsed time counters

Default: 1

See "Application lifespan (days)" in the Real-Time Workshop documentation.

Command line parameter

`LifeSpan`

Debugging	No impact
Traceability	No impact
Efficiency	Set to correct value
Safety precaution	`inf`

Parameter structure

Control how parameter data is generated for reusable subsystems

Hierarchical (default)
: Generate a separate header file, defining an independent parameter structure, for each subsystem that meets the following conditions:

- Subsystem's **Real-Time Workshop system code** parameter is set to `Reusable function`
- Subsystem does not violate any code reuse limitations
- Subsystem does not access parameters other than its own (such as parameters of the root-level model)

Each subsystem parameter structure is referenced as a substructure of the root-level parameter data structure, creating a structure hierarchy.

NonHierarchical
Generate a single, flat parameter data structure. Subsystem parameters are defined as fields within the structure. A nonhierarchical data structure can reduce compiler padding between word boundaries, producing more efficient compiled code.

Dependencies

Inline parameters must be enabled

Command line parameter

`InlinedParameterPlacement`

Recommended settings

Debugging	No impact
Traceability	Hierarchical
Efficiency	NonHierarchical
Safety precaution	No impact

More information

- "Nonvirtual Subsystem Code Generation Options"

Remove root level I/O zero initialization

Specify whether to generate initialization code for root-level inports and outports set to zero

Checked

Do not generate initialization code for root-level inports and outports set to zero.

Unchecked (default)

Generate initialization code for all root-level inports and outports. You should use the default

- To ensure that memory allocated for C MEX S-function wrappers is initialized to zero
- For safety critical applications that require that all internal and external data be initialized to zero

Command line parameter

`ZeroExternalMemoryAtStartup`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	Clear

Use memset to initialize floats and doubles to 0.0

Specify whether to generate code that explicitly initializes floating-point data to 0.0

Checked

Use `memset` to clear internal storage for floating-point data to integer bit pattern 0 (all bits 0), regardless of type. An example of a case for selecting this option is to gain compiler efficiency when the compiler

and target CPU both represent floating-point zero with the integer bit pattern 0.

Unchecked (default)
Generate extra code to explicitly initialize storage for data of types `float` and `double` to 0.0. The resulting code is slightly less efficient than code generated when you select the option.

You should not select this option if you need to ensure that memory allocated for C MEX S-function wrappers is initialized to zero.

Command line parameter

`InitFltsAndDblsToZero`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	No impact

Remove internal state zero initialization

Specify whether to generate initialization code for internal work structures, such as block states and block outputs, to zero

Checked
Do not generate code that initializes internal work structures to zero. An example of when you might select this option is to test the behavior of a design during warm boot—a restart without full system reinitialization.

Selecting this option does not guarantee that memory is in a known state each time the generated code begins execution. When you run a model or generated S-function multiple times, each run can produce a different answer.

If want to get the same answer on every run from a generated S-function, enter the command `clear SFcnNam` or `clear mex` in the MATLAB Command Window before each run.

Unchecked (default)
Generate code that initializes internal work structures to zero. You should use the default

- To ensure that memory allocated for C MEX S-function wrappers is initialized to zero
- For safety critical applications that require that all internal and external data be initialized to zero

Command line parameter

`ZeroInternalMemoryAtStartup`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	Clear

Optimize initialization code for model reference

Specify whether to generate initialization code for blocks that have states

Checked (default)
Suppress generation of initialization code for blocks that have states unless the blocks are in a system that can reset its states, such as an enabled subsystem. This results in more efficient code, but requires that you not refer to the model from a Model block that resides in a system that resets states. If you violate this constraint, Simulink reports an error, in which case you can disable this optimization.

Unchecked
Generate initialization code for all blocks that have states. You should disable this option if the current model includes a subsystem that resets

states, such as an enabled subsystem, and the model is referred to from another model with a Model block.

Dependencies

Must uncheck if model includes enabled subsystem and model is referred to from another model with Model block

Command line parameter

`OptimizeModelRefInitCode`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	No impact

Remove code that protects against division arithmetic exceptions

Specify whether to generate code that guards against division by zero for fixed-point data

Checked

Do not generate code that guards against division by zero for fixed-point data. When you select this option, simulation results and results from generated code might not be in bit-for-bit agreement.

Unchecked (default)

Generate code that guards against division by zero for fixed-point data.

Command line parameter

`NoFixptDivByZeroProtection`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	Clear

Real-Time Workshop (General)

Include hyperlinks to model

Include hyperlinks in generated HTML report that link code to corresponding blocks in model diagram

Checked
: Include hyperlinks in the generated HTML report that link code to corresponding blocks in the model diagram. The hyperlinks provide traceability for validating generated code against the source model.

Unchecked (default)
: Omit hyperlinks from the generated report. Disable this option to speed up code generation. For very large models (containing over 1000 blocks), generation of hyperlinks can be time consuming.

Dependencies

Automatically enabled and selected when you select **Generate HTML report**

Command line parameter

`IncludeHyperlinkInReport`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	Set

Ignore custom storage classes

Specify whether to apply or ignore custom storage classes

Checked

Ignore custom storage classes by treating data objects that have them as if their storage class attribute is set to `Auto`. Data objects with an `Auto` storage class do not interface with external code and are stored as local or shared variables or in a global data structure.

Unchecked (default)

Apply custom storage classes as specified. You must uncheck this option if the model defines data objects with custom storage classes.

Dependencies

- Uncheck before configuring data objects with custom storage classes.
- Must uncheck to enable module packaging features.
- Setting for top-level and referenced models must match.

Command line parameter

`IgnoreCustomStorageClasses`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

Comments

Simulink block descriptions

Specify whether to insert descriptions of blocks into generated code as comments

Checked

Include the following comments in the generated code for each block in the model, with the exception of virtual blocks and blocks removed due to block reduction:

- The block name at the start of the code, regardless of whether you select **Simulink block comments**
- Text specified in the **Description** field of each Block Parameter dialog box

The block names and descriptions can include international (non-US-ASCII) characters

Unchecked (default)

Suppress the generation of block name and description comments in the generated code.

Command line parameter

`InsertBlockDesc`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

Support for International (Non-US-ASCII) Characters in the Real-Time Workshop documentation

Simulink data object descriptions

Specify whether to insert descriptions of Simulink data objects into generated code as comments

Checked

Insert contents of **Description** field of Model Explorer Object Properties pane for each Simulink data object (signal, parameter, and bus objects) in the generated code as comments.

The descriptions can include international (non-US-ASCII) characters

Unchecked (default)

Suppress the generation of data object property descriptions as comments in the generated code.

Command line parameter

`SimulinkDataObjDesc`

Recommended settings

Debugging	Set
Traceability	Set

Efficiency	No impact
Safety precaution	No impact

Custom comments (MPT objects only)

Specify whether to include custom comments for MPT signal and parameter data objects in generated code

Checked
: Insert comments just above the identifiers for signal and parameter MPT objects in generated code. This option enables a **Custom comments function** field, which you use to specify an M-code or TLC function that contains your custom comments. For example, you might use this option to insert comments that document some or all of an object's property values.

Unchecked (default)
: Suppress the generation of custom comments for signal and parameter identifiers.

Dependencies

Requires that you include the comments in a function defined in an M-file or TLC file that you specify with **Custom comments function**.

Command line parameter

`EnableCustomComments`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

"Adding Custom Comments"

Custom comments function (MPT objects only)

Specify file that contains comments to be included in generated code for MPT signal and parameter data objects

Enter the name of the M-file or TLC file for the function that includes the comments to be inserted of your MPT signal and parameter objects. You can specify the file name directly or click **Browse** and search for a file.

Dependencies

Enabled with **Custom comments**

Command line parameter

`CustomCommentsFcn`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

"Adding Custom Comments"

Stateflow object descriptions

Specify whether to insert descriptions of Stateflow objects into generated code as comments

Checked
Insert the contents of the **Description** field of the Model Explorer Object Properties pane for each Stateflow object (states, charts, transitions, or graphical functions) into the generated code as comments. The comments appear just above the code generated for each object.

The descriptions can include international (non-US-ASCII) characters.

Unchecked (default)
Suppress the generation of comments for Stateflow objects.

Command line parameter

`SFDataObjDesc`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

Support for International (Non-US-ASCII) Characters in the Real-Time Workshop documentation

Requirements in block comments

Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments

Checked
Insert the requirement descriptions that you assign to Simulink blocks into the generated code as comments. Real-Time Workshop includes the requirement descriptions in the generated code in the following locations.

Model Element	Requirement Description Location
Model	In the main header file *model*.h
Nonvirtual subsystems	At the call site for the subsystem
Virtual subsystems	At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem has no nonvirtual parent, requirement descriptions are located in the main header file for the model, *model*.h.
Nonsubsystem blocks	In the generated code for the block

The requirement text can include international (non-US-ASCII) characters.

Unchecked (default)
Suppress the generation of comments for block requirement descriptions.

Command line parameter

`ReqsInCode`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	Set

More information

Including Requirements with Generated Code in the Simulink Verification and Validation documentation

Symbols

Global variables

Customize generated global variable identifiers

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens:

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.

Token	Description
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing.

Default: `$R$N$M`

Tips

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model.
- Where possible, increase the Maximum identifier length to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify`$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Command line parameter

`CustomSymbolStrGlobalVar`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	`$R$N$M`

More information

- “Specifying Identifier Formats”
- “Name Mangling”
- “Model Referencing Considerations”
- “Identifier Format Control Parameters Limitations”

Global types

Customize generated global type identifiers

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens:

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing.

Default: `$N$R$M`

Tips

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model.
- Where possible, increase the Maximum identifier length to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify`$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Command line parameter

`CustomSymbolStrType`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	`$N$R$M`

More information

- “Specifying Identifier Formats”
- “Name Mangling”
- “Model Referencing Considerations”
- “Identifier Format Control Parameters Limitations”

Field name of global types

Customize generated field names of global types

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens:

Token	Description
`$A`	Insert data type acronym (for example, `i32` for long integers) into signal and work vector identifiers.
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string `root_`. For blocks at the subsystem level, the tag is of the form `sN_`, where `N` is a unique system number assigned by Simulink.
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

Default: `$N$M`

Tips

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model.
- Where possible, increase the Maximum identifier length to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- The **Maximum identifier length** setting does not apply to type definitions.
- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Command line parameter

`CustomSymbolStrField`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	`$N$M`

More information

- “Specifying Identifier Formats”
- “Name Mangling”
- “Identifier Format Control Parameters Limitations”

Subsystem methods

Customize generated global type identifiers

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens:

Token	Description
`$F`	Insert method name (for example, `_Update` for update method). Empty for Stateflow functions.
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string `root_`. For blocks at the subsystem level, the tag is of the form `sN_`, where `N` is a unique system number assigned by Simulink. Empty for Stateflow functions.
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing.

Default: `$R$N$M$F`

Tips

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model.
- Where possible, increase the Maximum identifier length to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.

- If you specify$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the $R and $M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, typedef statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify $R, the code generator includes the model name in the typedef.
- This option does not affect objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Command line parameter

CustomSymbolStrFcn

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	RNMF

More information

- “Specifying Identifier Formats”
- “Name Mangling”
- “Model Referencing Considerations”
- “Identifier Format Control Parameters Limitations”

Local temporary variables

Customize generated local temporary variable identifiers

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens:

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing.

Default: `$N$M`

Tips

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model.
- Where possible, increase the Maximum identifier length to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify`$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Command line parameter

`CustomSymbolStrTmpVar`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	`$N$M`

More information

- “Specifying Identifier Formats”
- “Name Mangling”
- “Model Referencing Considerations”
- “Identifier Format Control Parameters Limitations”

Local block output variables

Customize generated local block output variable identifiers

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens:

Token	Description
`$A`	Insert data type acronym (for example, `i32` for long integers) into signal and work vector identifiers.

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

Default: `rtb_$N$M`

Tips

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model.
- Where possible, increase the Maximum identifier length to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Command line parameter

`CustomSymbolStrBlkIO`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	`rtb_$N$M`

More information

- “Specifying Identifier Formats”
- “Name Mangling”
- “Identifier Format Control Parameters Limitations”

Constant macros

Customize generated constant macro identifiers

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens:

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing.

Default: `$R$N$M`

Tips

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, Gain1, Gain2...) when there are many blocks of the same type in the model.
- Where possible, increase the Maximum identifier length to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.

- If you specify`$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Command line parameter

`CustomSymbolStrMacro`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	`$R$N$M`

More information

- “Specifying Identifier Formats”
- “Name Mangling”
- “Model Referencing Considerations”
- “Identifier Format Control Parameters Limitations”

Minimum mangle length

Increase the minimum number of characters used for generating name mangling strings that help avoid name collisions

Specify an integer value that indicates the minimum number of characters the code generator is to use when generating a name mangling string. As necessary, the minimum value automatically increases during code generation as a function of the number of collisions. A larger value reduces the chance of identifier disturbance when you modify the model.

Default: 1

Tips

- Minimize disturbance to the generated code during development, by specifying a value of 4. This value is conservative and safe; it allows for over 1.5 million collisions for a particular identifier before the mangle length is increases.
- Set the value to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

Command line parameter

`MangleLength`

Recommended settings

Debugging	No impact
Traceability	1
Efficiency	No impact
Safety precaution	4

More information

- “Name Mangling”
- “Traceability”
- “Minimizing Name Mangling”

Generate scalar inlined parameter as

Control how scalar inlined parameter values are expressed in the generated code

Literals (default)
: Generate scalar inlined parameters as numeric constants. This setting can help with debugging TLC code, as it makes it easy to search for parameter values in the generated code.

Macros
: Generate scalar inlined parameters as variables with `#define` macros. This setting makes generated code more readable.

Dependency

Enabled with **Inline parameters**

Command line parameter

`InlinedPrmAccess`

Recommended settings

Debugging	No impact
Traceability	Macros
Efficiency	Literals
Safety precaution	No impact

Signal naming

Specify rule for naming signals in generated code

None (default)
: Make no change to signal names when creating corresponding identifiers in generated code. Signal identifiers in the generated code match the signal names that appear in the model.

Force upper case
Use all uppercase characters when creating identifiers for signal names in the generated code.

Force lower case
Use all lowercase characters when creating identifiers for signal names in the generated code.

Custom M-function
Use the M-file function specified with the **M-function** parameter to create identifiers for signal names in the generated code.

Dependencies

- Enables **M-function** parameter when set to `Custom M-function`
- Must be the same for top-level and referenced models

Command line parameter

`SignalNamingRule`

Recommended settings

Debugging	No impact
Traceability	Force uppercase
Efficiency	No impact
Safety precaution	No impact

More information

- "Applying Naming Rules to Identifiers Globally"
- MATLAB Programming in the MATLAB documentation

Parameter naming

Specify rule for naming parameters in generated code

None (default)
: Make no change to parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case
: Use all uppercase characters when creating identifiers for parameter names in the generated code.

Force lower case
: Use all lowercase characters when creating identifiers for parameter names in the generated code.

Custom M-function
: Use the M-file function specified with the **M-function** parameter to create identifiers for parameter names in the generated code.

Dependencies

- Enables **M-function** parameter when set to `Custom M-function`
- Must be the same for top-level and referenced models

Command line parameter

`ParamNamingRule`

Recommended settings

Debugging	No impact
Traceability	Force uppercase
Efficiency	No impact
Safety precaution	No impact

More information

- “Applying Naming Rules to Identifiers Globally”
- MATLAB Programming in the MATLAB documentation

#define naming

Specify rule for naming `#define` parameters (defined with storage class `Define (Custom)`) in generated code

None (default)
: Make no change to `#define` parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case
: Use all uppercase characters when creating identifiers for `#define` parameter names in the generated code.

Force lower case
: Use all lowercase characters when creating identifiers for `#define` parameter names in the generated code.

Custom M-function
: Use the M-file function specified with the **M-function** parameter to create identifiers for `#define` parameter names in the generated code.

Dependencies

- Enables **M-function** parameter when set to `Custom M-function`
- Must be the same for top-level and referenced models

Command line parameter

`DefineNamingRule`

Recommended settings

Debugging	No impact
Traceability	Force uppercase
Efficiency	No impact
Safety precaution	No impact

More information

- “Applying Naming Rules to Identifiers Globally”
- MATLAB Programming in the MATLAB documentation

M-function

Specify rule for naming identifiers in generated code

Enter the name of an M-file that contains the naming rule to be applied to signal, parameter, or `#define` parameter identifiers in generated code. Examples of rules you might program in such an M-file function include

- Remove underscore characters from signal names
- Add an underscore before uppercase characters in parameter names
- Make all identifiers uppercase in generated code

Dependencies

- Enabled with the following parameters:

  Signal naming
  Parameter naming
  #define naming

- M-file must be in the MATLAB path
- Must be the same for top-level and referenced models

Command line parameter

`DefineNamingFcn`

Recommended settings

Debugging	No impact
Traceability	Force uppercase

Efficiency	No impact
Safety precaution	No impact

More information

- “Applying Naming Rules to Identifiers Globally”
- MATLAB Programming in the MATLAB documentation

Interface

Support floating-point numbers

Specify whether to generate floating-point data and operations

Checked (default)
: Generate floating-point data and operations.

Unchecked
: Generate pure integer code. If you uncheck this option, an error occurs if the code generator encounters floating-point data or expressions. The error message reports offending blocks and parameters.

Dependency

Setting for top-level model and referenced models must match

Command line parameter

`PurelyIntegerCode`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Clear for integer only
Safety precaution	No impact

Support absolute time

Specify whether to generate and maintain integer counters for absolute and elapsed time values

Checked (default)

Generate and maintain integer counters for blocks that require absolute or elapsed time values. Absolute time is the time from the start of program execution to the present time. An example of elapsed time is time elapsed between two trigger events.

If you select this option and the model does not include blocks that use time values, the target does not generate the counters.

Unchecked

Do not generate integer counters to represent absolute or elapsed time values. If you do not select this option and the model includes blocks that require absolute or elapsed time values, an error occurs during code generation.

Dependency

Must enable if model includes blocks that require absolute or elapsed time values

Command line parameter

`SupportAbsoluteTime`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Clear
Safety precaution	Clear

More information

Timing Services

Support non-finite numbers

Specify whether to generate nonfinite data and operations

Checked (default)
: Generate nonfinite data (for example, `NaN` and `Inf`) and related operations.

Unchecked
: Do not generate nonfinite data and operations. If you uncheck this option, an error occurs if the code generator encounters nonfinite data or expressions. The error message reports offending blocks and parameters.

Dependency

- Enabled by **Support floating-point numbers**
- Setting for top-level model and referenced models must match

Command line parameter

`SupportNonFinite`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Clear
Safety precaution	Clear

Support continuous time

Specify whether to generate code for blocks that use continuous time

Checked
Generate code for blocks that use continuous time.

Unchecked(default)
Do not generate code for blocks that use continuous time. If you do not select this option and the model includes blocks that use continuous time, an error occurs during code generation.

Dependency

- Must enable if model includes blocks that require absolute or elapsed time values
- Must disable if generating an S-function wrapper for an ERT target; the code generator does not support continuous time for this target scenario

Command line parameter

`SupportContinuousTime`

Recommended settings

Debugging	No impact
Traceability	No impact

Efficiency	Clear
Safety precaution	Clear

More information

- “Support for Continuous Time Blocks, Continuous Solvers, and Stop Time”
- “Automatic S-Function Wrapper Generation”

Support complex numbers

Specify whether to generate complex data and operations

Checked (default)
: Generate complex numbers and related operations.

Unchecked
: Do not generate complex data and related operations. If you uncheck this option, an error occurs if the code generator encounters complex data or expressions. The error message reports offending blocks and parameters.

Dependency

- Enabled by **Support floating-point numbers**
- Setting for top-level model and referenced models must match

Command line parameter

`SupportComplex`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Clear for real only
Safety precaution	No impact

Support non-inlined S-functions

Specify whether to generate code for non-inlined S-functions

Checked
Generate code for non-inlined S-functions.

Unchecked(default)
Do not generate code for non-inlined S-functions. If you do not select this option and the model includes a C MEX S-function that does not have a corresponding TLC implementation (for inlining), an error occurs during the build process.

Note that inlining S-functions is highly advantageous in production code generation, for example for implementing device drivers. In such cases, you should uncheck this option to enforce use of inlined S-functions for code generation.

Dependency

Automatically enables **Support for floating-point numbers** and **Support for non-finite numbers**, which are required for this option

Command line parameter

`SupportNonInlinedSFcns`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Clear
Safety precaution	Clear

More information

“Automatic S-Function Wrapper Generation”

GRT compatible call interface

Specify whether to generate model function calls compatible with main program module of GRT target

Checked
: Generate model function calls that are compatible with the main program module of the GRT target (`grt_main.c` or `grt_main.cpp`).

  This option provides a quick way to use ERT target features with a GRT-based custom target that has a main program module based on `grt_main.c` or `grt_main.cpp`.

Unchecked(default)
: Disable the GRT compatible call interface.

Dependencies

When enabled

- Must enable **MAT-file logging**
- Must disable **Single output/update function** and **Suppress error status in real-time model data structure**
- The following are unsupported
  - Data type replacement
  - Nonvirtual subsystem option **Function with separate data**

Command line parameter

`GRTInterface`

Recommended settings

Debugging	No impact
Traceability	Clear
Efficiency	Clear
Safety precaution	Clear

More information

"Support for Continuous Time Blocks, Continuous Solvers, and Stop Time"

Single output/update function

Specify whether to generate *model*_step function

Checked (default)

Generate the model's *model*_step function. This function contains the output and update function code for all blocks in the model and is be called by rt_OneStep to execute processing for one clock period of the model at interrupt level.

Unchecked

Do not combine output and update function code for model blocks in a single function. Generate the code in *model*.output and *model*.update.

Dependency

Must disable

- **GRT compatible call interface**
- **Minimize algebraic loop occurrences**

Command line parameter

CombineOutputUpdateFcns

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	Set
Safety precaution	Set

More information

"rt_OneStep"

Terminate function required

Specify whether to generate *model*_terminate function

Checked (default)
Generate a *model*_terminate function. This function contains all model termination code and should be called as part of system shutdown.

Unchecked
Do not generate a *model*.terminate function. Suppress the generation of this function if you designed your application to run indefinitely and does not require a terminate function.

Dependency

Setting for top-level and referenced models must match

Command line parameter

`IncludeMdlTerminateFcn`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Clear
Safety precaution	Clear

More information

`model_terminate`

Generate reusable code

Specify whether to generate reusable, reentrant code

Checked
Generate reusable, multi-instance code that is reentrant. The code generator passes model data structures (root-level inputs and outputs,

block states, parameters, and external outputs) in, by reference, as arguments to *model*_step and other the model entry point functions. The data structures are also exported with *model*.h. For efficiency, the code generator passes in only data structures that are used. Therefore, when you enable this option, the argument lists generated for the entry point functions vary according to model requirements.

Unchecked (default)
Do not generate reusable code. Model data structures are statically allocated and accessed by model entry point functions directly in the model code.

Tips

- Entry points are exported with *model*.h. To call the entry-point functions from hand-written code, add an `#include model.h` directive to the code. If this option is enabled, you must examine the generated code to determine the calling interface required for these functions.
- When this option is enabled, the code generator generates a pointer to the real-time model object (*model*_M).
- In some cases, when this option is enabled, the code generator might generate code that compiles but is not reentrant. For example, if any signal, DWork structure, or parameter data has a storage class other than Auto, global data structures are generated.

Dependencies

- Enables **Reusable code error diagnostic** and **Pass root-level I/O as** options
- Must disable this option if you are using
  - The static ert_main.c module, rather than generating a main program
  - The model_step function prototype control capability
  - The subsystem parameter **Function with separate data**
  - A subsystem that
    - Multiple ports that share the same source

  - A port used by multiple instances has different sample times, data types, complexity, frame status, or dimension across the instances
  - Has output marked as a global signal
  - For each instance contains identical blocks with different names or parameter settings
- Has no effect on code generated for function-call subsystems

Command line parameter

`MultiInstanceERTCode`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set for single instance
Safety precaution	No impact

More information

- “Model Entry Points”
- “Nonvirtual Subsystem Code Generation”
- “Code Reuse Limitations”
- “Determining Why Subsystem Code Is Not Reused”
- “Writing S-Functions That Support Code Reuse”
- “Static Main Program Module”
- “Controlling model_step Function Prototypes”
- “Nonvirtual Subsystem Modular Function Code Generation”
- “Exporting Function-Call Subsystems”
- `model_step`

Reusable code error diagnostic

Select the severity level for diagnostics displayed when model violates requirements for generating reusable code

None
Proceed with build without displaying a diagnostic message.

Warning
Proceed with build after displaying a warning message.

Error (default)
Abort build after displaying an error message.

Under certain conditions, Real-Time Workshop Embedded Coder might

- Generate code that compiles but is not reentrant. For example, if signal, DWork structure, or parameter data has a storage class other than `Auto`, global data structures are generated.
- Be unable to generate valid and compilable code. For example, if the model contains an S-function that is not code-reuse compliant or a subsystem triggered by a wide function-call trigger, the coder generates invalid code, displays an error message, and terminates the build.

Dependency

Enabled with the **Generate reusable code** option

Command line parameter

`MultiInstanceErrorCode`

Recommended settings

Debugging	`Warning` or `Error`
Traceability	No impact
Efficiency	`None`
Safety precaution	No impact

More information

- “Model Entry Points”
- “Nonvirtual Subsystem Code Generation”
- “Code Reuse Limitations”
- “Determining Why Subsystem Code Is Not Reused”
- “Nonvirtual Subsystem Modular Function Code Generation”

Pass root-level I/O as

Control how root-level model input and output are passed to *model*_step function

Individual arguments (default)
: Pass each root-level model input and output value to *model*_step as a separate argument.

Structure reference
: Pack all root-level model input into a `struct` and pass `struct` to *model*_step as an argument. Similarly, pack root-level model output into a second `struct` and pass it to *model*_step.

Dependency

Enabled with the **Generate reusable code** option

Command line parameter

`RootIOFormat`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Model Entry Points”
- “Nonvirtual Subsystem Code Generation”
- “Nonvirtual Subsystem Modular Function Code Generation”
- `model_step`

Suppress error status in real-time model data structure

Specify whether to log or monitor error status

If you do not need to log or monitor error status in your application, select the Suppress error status in real-time model data structure option. This further reduces memory usage. Selecting this option may also cause rtModel to disappear completely from the generated code.

Checked
: Do not include the error status field the in generated real-time model data structure `rtModel`. Enabling this option reduces memory usage.

  Note that enabling this option can cause the code generator to completely omit the `rtModel`data structure from generated code.

Unchecked (default)
: Include an error status field in the generated real-time model data structure `rtModel`. You can use available macros to monitor the field for or set it with error message data.

Dependencies

- Disable this option if you select **MAT-file logging**.
- Setting of this option for multiple integrated models must match to avoid unexpected application behavior. For example, if select the option for one model but not in another, an error status might not get registered by the integrated application.

Command line parameter

`SuppressErrorStatus`

Recommended settings

Debugging	Clear
Traceability	No impact
Efficiency	Set
Safety precaution	Set

More information

"rtModel Accessor Macros"

Configure Functions

Open Model Step Functions dialog box to configure *`model`*`_step` function prototype

Use the **Configure Functions** button to open the Model Step Functions dialog box. This dialog box provides a way for you to specify whether the code generator is to use a default *`model`*`_step` function prototype or a model-specific C prototype. Based on your selection, you can preview and modify the function prototype.

More information

- "Controlling model_step Function Prototypes"
- `model_step`
- "Model Step Functions Dialog Box"

Create Simulink (S-Function) block

Specify whether to generate an S-function block

Checked

Generate an S-function block to represent the model or subsystem. The coder generates an inlined C or C++ MEX S-function wrapper that calls existing hand-written code or code previously generated by Real-Time Workshop from within Simulink. S-function wrappers provide a standard interface between Simulink and externally written code, allowing you to integrate your code into a model with minimal modification.

When this option is enable, Real-Time Workshop

1. Generates the S-function wrapper file *`model`*`_sf.c` (or `.cpp`) and places it in the build directory
2. Builds the MEX-file *`model`*`_sf.`*`mexext`* and places it in your working directory
3. Creates and opens an untitled model containing the generated S-Function block

Unchecked (default)

Do not generate an S-function block.

Command line parameter

`GenerateErtSFunction`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Automatic S-Function Wrapper Generation”
- “Techniques for Exporting Function-Call Subsystems”
- “Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes”

Enable portable word sizes

Specify whether to allow portability across host and target processors that support different word sizes

Checked
: Generate conditional processing macros that support compilation of generated code on a processor that supports a different word size than the target processor on which production code is intended to run (for example, a 32–bit host and a 16-bit target. This allows you to use the same generated code for both software-in-the-loop (SIL) testing on the host platform and production deployment on the target platform.

Unchecked (default)
: Do not generated code that supports compilation of generated code on processors that support different word sizes.

Dependencies

- Enable **Create Simulink (S-Function) block**
- Set **Emulation hardware** on the **Hardware Implementation** pane to `None`

Command line parameter

`PortableWordSizes`

Recommended settings

Debugging
Traceability

Efficiency

Safety precaution

More information

"Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes""Tips for Optimizing the Generated Code"

MAT-file logging

Specify whether to enable MAT-file logging

Checked
Enable MAT-file logging. When you enable this option, the code generator saves system states, output, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) *`model`*`.mat`, where *`model`* is the name of your model.

Unchecked (default)
Disable MAT-file logging. Disabling this option has the following benefits

- Eliminates overhead associated with supporting a file system, which typically is not needed for embedded applications
- Eliminates extra code and memory usage required to initialize, update, and clean up logging variables
- Under certain conditions, eliminates code and storage associated with root output ports
- Omits the comparison between the current time and stop time in the *`model`*`_step`, allowing the generated program to run indefinitely, regardless of the stop time setting

Dependency

- When selected, **Support floating-point numbers** and **Support non-finite numbers** are also selected automatically.

- If you select **GRT compatible call interface**, you must select this option and deselect **Suppress error status in real-time model data structure**.
- Deselect this option if you are using exported function calls.

Command line parameter

`MatFileLogging`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	Clear
Safety precaution	Clear

More information

- "Data Logging"
- "Using Virtualized Output Ports Optimization"

Code Style

Parentheses level

Specify parenthesization style for generated code

Minimum
Insert parentheses only where required by ANSI C or needed to override default precedence. For example:

```
isZero = var == 0;
if (isZero == 1 && (value < 3.7 ||value > 9.27)) {
   /* code */
   }
```

Nominal (default)
Insert parentheses in a way that compromises between readability and visual complexity. The exact definition can change between releases.

Maximum
Include parentheses everywhere needed to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA requirements. For example:

```
isZero = (var == 0);
if ((isZero == 1) && ((value < 3.7) || (value > 9.27))) {
   /* code */
   }
```

Command line parameter

`ParenthesesLevel`

Recommended settings

Debugging	
Traceability	
Efficiency	
Safety precaution	`Maximum`

More information

"Controlling Parenthesization"

Preserve operand order in expression

Specify whether to preserve order of operands in expressions

Checked

Preserve the expression order specified in the model. Select this option to increase readability of the code or for code traceability purposes.

```
A*(B+C)
```

Unchecked (default)

Optimize efficiency of code for nonoptimized compilers by reordering commutable operands to make expressions left-recursive. For example:

```
(B+C)*A
```

Command line parameter

`PreserveExpressionOrder`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	Clear
Safety precaution	Set

Preserve condition expression in if statement

Specify whether to preserve empty primary condition expressions in `if` statements

Checked

Preserve empty primary condition expressions in `if` statements, such as the following, to increase the readability of the code or for code traceability purposes.

```
if expression1
else
       statements2;
end
```

Unchecked (default)

Optimize empty primary condition expressions in `if` statements by negating them. For example, consider the following `if` statement:

```
if expression1
else
       statements2;
end
```

Be default, the code generator negates this statement as follows:

```
if ~expression1
        statements2;
end
```

Command line parameter

`PreserveIfCondition`

Recommended settings

Debugging	Set
Traceability	Set

Efficiency	Clear
Safety precaution	Set

Templates

Code templates: Source file (*.c) template

Specify code generation template (CGT) file to use when generating source code file

You can use a CGT file to define the top-level organization and formatting of generated source code (`.c` or `.cpp`) files. The CGT file must be located on the MATLAB path.

Default:
`matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`

Command line parameter

`ERTSrcFileBannerTemplate`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Selecting and Defining Templates”
- “Custom File Processing”

Code templates: Header file (*.h) template

Specify code generation template (CGT) file to use when generating code header file

You can use a CGT file to define the top-level organization and formatting of generated header files (`.h`). The CGT file must be located on the MATLAB path.

Default:
`matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`

Command line parameter

`ERTHdrFileBannerTemplate`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Selecting and Defining Templates”
- “Custom File Processing”

Data templates: Source file (*.c) template

Specify code generation template (CGT) file to use when generating data source file

You can use a CGT file to define the top-level organization and formatting of generated data source files (`.c` or `.cpp`) that contain definitions of variables of global scope. The CGT file must be located on the MATLAB path.

Default:
`matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`

Command line parameter
`ERTDataSrcFileTemplate`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Selecting and Defining Templates”
- “Custom File Processing”

Data templates: Header file (*.h) template
Specify code generation template (CGT) file to use when generating data header file

You can use a CGT file to define the top-level organization and formatting of generated data header files (`.h`) that contain declarations of variables of global scope. The CGT file must be located on the MATLAB path.

Default:
`matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`

Command line parameter

`ERTDataHdrFileTemplate`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "Selecting and Defining Templates"
- "Custom File Processing"

File customization template

Specify custom file processing (CFP) template file to use when generating code

You can use a CFP template file to customize generated code. A CFP template file is a TLC file that organizes types of code (for example, includes, `typedefs`, and functions) into sections,. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call a code template API to emit the buffered code into specified sections of generated source and header files. The CFP template file must be located on the MATLAB path.

Default:

`matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`

Command line parameter

`ERTCustomFileTemplate`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "Selecting and Defining Templates"
- "Custom File Processing"
- *Real-Time Workshop Target Language Compiler*

Generate an example main program

Control whether to generate example main program for model

Checked (default)
: Generate example main program,`ert_main.c` (or `.cpp`) . The file includes:

  - The `main()` function for the generated program
  - Task scheduling code that determines how and when block computations execute on each time step of the model

  The operation of the main program and the scheduling algorithm employed depend primarily upon whether your model is single-rate or multirate, and also upon your model's solver mode (`SingleTasking` or `MultiTasking`).

Unchecked
: Provide a static version of the file `ert_main.c` as a basis for custom modifications (*`matlabroot`*`/rtw/c/ert/ert_main.c`). You can use this file as a template for developing embedded applications.

Tips

- After you generate and customize the main program, disable this option to prevent regenerating the main module and overwriting your customized version.
- You can use a custom file processing (CFP) template file to override normal main program generation, and generate a main program module customized for your target environment.
- If you disable this option, the coder generates slightly different rate grouping code to maintain compatibility with an older static `ert_main.c` module.

Dependencies

- Enables **Target operating system**
- You must enable this option and select `VxWorksExample` for **Target operation system** if you use VxWorks library blocks

Command line parameter

`GenerateSampleERTMain`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Generating the Main Program Module”
- “Static Main Program Module”
- “Custom File Processing”

Target operating system

Specify target operating system to use when generating model-specific example main program module

BareBoardExample (default)
: Generate a bareboard main program designed to run under control of a real-time clock, without a real-time operating system.

VxWorksExample
: Generate a fully commented example showing how to deploy the code under the VxWorks real-time operating system.

Dependencies

- Enabled when you select **Generate an example main program**
- Must be the same for top-level and referenced models.

Command line parameter

`TargetOS`

More information

- “Generating the Main Program Module”
- “Static Main Program Module”
- “Custom File Processing”

Data Placement

Data definition

Specify where definitions of variables of global scope are to be defined

Auto (default)
Let the code generator determine where the definitions should be located

Data defined in source file
Place definitions in `.c` source files where functions are located. The code generator places the definitions in one or more function `.c` files , depending on the number of function source files and the file partitioning previously selected in Simulink for the model.

Data defined in a single separate source file
Place definitions in the source file specified in the **Data definition filename** field. The code generator organizes and formats the definitions based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Templates** pane.

Dependencies

- Applies to data with custom storage classes only
- Enables **Data definition filename**

Command line parameter

`GlobalDataDefinition`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

- "Overview of Data Placement"
- "Managing File Placement of Data Definitions and Declarations"
- "Data Placement Rules and Effects"

Data definition filename

Specify name of the file that is to contain data definitions

The code generator organizes and formats the data definitions in the specified file based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Templates** pane.

If you specify C++ as the target language, omit the `.cpp` extension. The code generator will generate the correct file and add the extension `.cpp`.

Default: `global.cor` `global.cpp`

Command line parameter

`DataDefinitionFile`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

- "Selecting and Defining Templates"
- "Custom File Processing"

Data declaration

Specify where `extern`, `typedef`, and `#define` statements are to be declared

Auto (default)
: Let the code generator determine where the declarations should be located

Data declared in source file
: Place declarations in `.c` source files where functions are located. The data header template file is not used. The code generator places the declarations in one or more function `.c` files , depending on the number of function source files and the file partitioning previously selected in Simulink for the model.

Data defined in a single separate source file
: Place declarations in the data header file specified in the **Data declaration filename** field. The code generator organizes and formats the declarations based on the data header template specified by the **header file (*.h) template** parameter in the data section of the **Templates** pane.

Dependencies

- Applies to data with custom storage classes only

- Enables **Data declaration filename**

Command line parameter

`GlobalDataReference`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

- “Overview of Data Placement”
- “Managing File Placement of Data Definitions and Declarations”
- “Data Placement Rules and Effects”

Data declaration filename

Specify name of the file that is to contain data declarations

The code generator organizes and formats the data declarations in the specified file based on the data header template specified by the **Header file (*.h) template** parameter in the data section of the **Templates** pane.

If you specify C++ as the target language, omit the `.cpp` extension. The code generator will generate the correct file and add the extension `.cpp`.

Default: `global.cor` `global.cpp`

Command line parameter

`DataDeclarationFile`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

- "Selecting and Defining Templates"
- "Custom File Processing"

#include file delimiter

Specify type of `#include` file delimiter to use in generated code

`Auto` (default)
: Let the code generator choose the `#include` file delimiter

`#include  header.h`
: Use double quote (" ") characters to delimit file names in `#include` statements.

`#include <header.h>`
: Use angle brackets (< >) to delimit file names in `#include` statements.

Dependency

The delimiter format that you use when specifying parameter and signal object property values overrides what you set for this option.

Command line parameter

`IncludeFileDelimiter`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

Module naming

Specify whether to name module that owns the model

Not specified (default)
: Let the code generator determine the module name.

Same as model
: Use the name of the model for the module name.

User specified
: Use the module name specified for **Module name** parameter for the module name.

Command line parameter

`ModuleNamingRule`

Dependency

- Enables **Module name** when set to `User specified`.
- Use with the data object property **Owner** to specify module ownership.
- Setting must be the same for top-level and referenced models.

Recommended settings

Debugging	No impact
Traceability	Set

Efficiency	No impact
Safety precaution	No impact

More information

- “Overview of Data Placement”
- “Ownership Settings”

Module name

Specify name of module that is to own model

Specify a module name according to ANSI C/C++ conventions for naming identifiers.

Dependencies

- Enabled when you select `User specified` for **Module naming**.

Command line parameter

`ModuleName`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

- “Overview of Data Placement”
- “Ownership Settings”

Signal display level

Specify persistence level for all MPT signal data objects

Specify an integer value indicating the persistence level for all MPT signal data objects. This value indicates the level at which to declare signal data objects as global data in the generated code. The persistence level allows you to make intermediate variables global during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value you can specify for a specific MPT signal data object in the Model Explorer signal properties dialog.

Default: 10

Dependency

Setting must be the same for top-level and referenced models.

Command line parameter

`SignalDisplayLevel`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

“Selecting Persistence Level for Signals and Parameters”

Parameter tune level

Specify persistence level for all MPT parameter data objects

Specify an integer value indicating the persistence level for all MPT parameter data objects. This value indicates the level at which to declare parameter data objects as tunable global data in the generated code. The persistence level allows you to make intermediate variables global and tunable during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value you can specify for a specific MPT parameter data object in the Model Explorer parameter properties dialog.

Default: 10

Dependency

Setting must be the same for top-level and referenced models.

Command line parameter

`ParamTuneLevel`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

“Selecting Persistence Level for Signals and Parameters”

Data Type Replacement

Replace data type names in the generated code

Specify whether to replace built-in data type names with user-defined data type names in generated code

Checked

Display the **Data type names** table. The table provides a way for you to replace the names of built-in data types used in generated code. This mechanism can be particularly useful for generating code that adheres to application or site data type naming standards.

You can choose to specify new data type names for some or all Simulink built-in data types listed in the table. For each replacement data type name that you specify

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, and `boolean`, the replacement data type's `BaseType` must match the built-in data type.
- For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for `int` or `char` on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Unchecked (default)

Use Real-Time Workshop names for built-in Simulink data types in generated code.

Dependencies

Enables the following parameters:

double Replacement Name
single Replacement Name
int32 Replacement Name
int16 Replacement Name
int8 Replacement Name
uint32 Replacement Name
uint16 Replacement Name
uint8 Replacement Name
boolean Replacement Name
int Replacement Name
uint Replacement Name
char Replacement Name

Command line parameter

`EnableUserReplacementTypes`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	Clear

More information

“Replacing Built-In Data Type Names in Generated Code”

Replacement Name

Specify names to use for built-in Simulink data types in generated code

Specify strings that the code generator is to use as names for built-in Simulink data types .

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, and `boolean`, the replacement data type's `BaseType` must match the built-in data type.
- For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for `int` or `char` on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

Enabled when you select **Replace data type names in the generated code**.

Command line parameter

`ReplacementTypes`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	No impact
Safety precaution	Clear

More information

"Replacing Built-In Data Type Names in Generated Code"

Memory Sections

Package

Specify package that contains memory sections you want to apply to model-level functions and internal data

---None--- (Default)
No memory sections to apply.

Simulink
Apply built-in `Simulink` package.

mpt
Apply built-in `mpt` package.

If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Dependency

To include user-defined packages in the list of packages displayed, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

Command line parameter

`MemSecPackage`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Memory Sections"

Refresh package list

Add user-defined packages that are on search path to list of packages displayed by **Packages**

If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

More information

"Memory Sections"

Initialize/Terminate

Specify whether to apply memory section to Initialize/Start and Terminate functions

Default

Suppress the use of a memory section for Initialize/Start and Terminate functions.

memory-section-name

Apply memory section to Initialize/Start and Terminate functions.

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Command line parameter

`MemSecFuncInitTerm`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Memory Sections"

Execution

Specify whether to apply memory section to execution functions

Default

Suppress the use of a memory section for Step, Run-time initialization, Derivative, Enable, and Disable functions.

memory-section-name

Apply memory section to Step, Run-time initialization, Derivative, Enable, and Disable functions.

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Command line parameter

`MemSecFuncExecute`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Memory Sections"

Constants

Specify whether to apply memory section to constants

Applies to

Data Definition	Data Purpose
*model*_cP	Constant parameters
*model*_cB	Constant block I/O
*model*_Z	Zero representation

Default

Suppress the use of a memory section for constants.

memory-section-name

Apply memory section to constants.

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Command line parameter

`MemSecDataConstants`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Memory Sections"

Inputs/Outputs

Specify whether to apply memory section to root input and output

Applies to

Data Definition	Data Purpose
*model*_U	Root-level input
*model*_Y	Root-level output

Default

Suppress the use of a memory section for root-level input and output.

memory-section-name

Apply memory section for root-level input and output.

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Command line parameter

`MemSecDataIO`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Memory Sections"

Internal data

Specify whether to apply memory section to internal data

Applies to

Data Definition	Data Purpose
*model*_B	Block I/O
*model*_D	Dwork vectors
*model*_M	Run-time model
*model*_Zero	Zero-crossings

Default

Suppress the use of a memory section for internal data.

memory-section-name

Apply memory section for internal data.

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Command line parameter

`MemSecDataInternal`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Memory Sections"

Parameters

Specify whether to apply memory section to parameters

Applies to

Data Definition	Data Purpose
`model``_P`	Parameters

Default

Suppress the use of a memory section for parameters.

memory-section-name

Apply memory section for parameters.

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Command line parameter

`MemSecDataParameters`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Memory Sections"

Validation results

Display results of memory section validation

Real-Time Workshop checks and reports whether the currently chosen package is on the MATLAB path and that the selected memory sections exist inside the package. Then

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Index

A

B

C

D

E

F

G

I

M

O

P

R

S

T