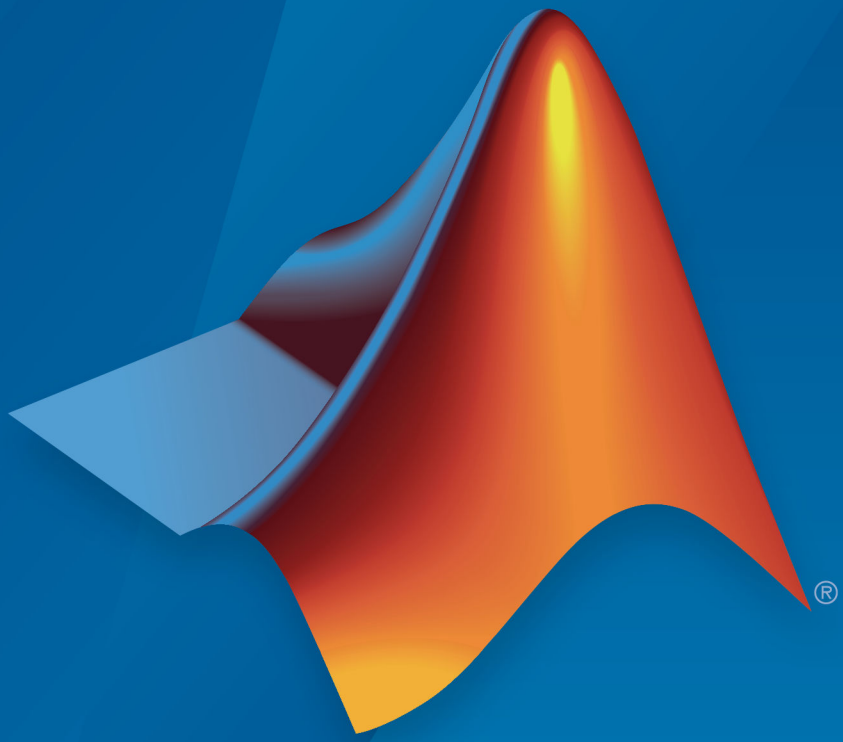
Embedded Coder®

Reference



MATLAB® & SIMULINK®

R2019a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Embedded Coder® Reference

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.3 (Release 2012b)
March 2013	Online only	Revised for Version 6.4 (Release 2013a)
September 2013	Online only	Revised for Version 6.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.6 (Release 2014a)
October 2014	Online only	Revised for Version 6.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.8 (Release 2015a)
September 2015	Online only	Revised for Version 6.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.10 (Release 2016a)
September 2016	Online only	Revised for Version 6.11 (Release 2016b)
March 2017	Online only	Revised for Version 6.12 (Release 2017a)
September 2017	Online only	Revised for Version 6.13 (Release 2017b)
March 2018	Online only	Revised for Version 7.0 (Release 2018a)
September 2018	Online only	Revised for Version 7.1 (Release 2018b)
March 2019	Online only	Revised for Version 7.2 (Release 2019a)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Contents

6 Code Generation Parameters: AUTOSAR

8 Code Generation Parameters: Code Style

9 Code Generation Parameters: Data Type Replacement

10 Memory Sections Parameters on the Code Generation Pane

11 Code Generation Parameters: Templates

12 Code Generation Parameters: Verification

13 Configuration Parameters

14 Parameters for Creating Protected Models

15 Model Advisor Checks

16 Tools in Embedded Coder—Alphabetical List

17 C/C++ Functions That Support Symbolic Dimensions for Simulink Function Blocks

1

Functions in Embedded Coder—Alphabetical List

activateConfigSet

Class: cgv.CGV
Package: cgv

Activate configuration set of model

Syntax

`cgvObj.activateConfigSet(configSetName)`

Description

`cgvObj.activateConfigSet(configSetName)` specifies the active configuration set for the model, only while the model is executed by *cgvObj*. *cgvObj* is a handle to a `cgv.CGV` object. `configSetName` is the name of a configuration set object, `ConfigSet`, which already exists in the model. The original configuration set for the model is restored after execution of the `cgv.CGV` object.

Examples

Before calling `cgv.CGV.run` on a `cgv.CGV` object for a model, the model must already contain the named configuration set. After creating the `cgv.CGV` object for a model, you can use `cgv.CGV.activateConfigSet` to activate a configuration set in the model when the `cgv.CGV` object simulates the model.

```
configObj = Simulink.ConfigSet;
attachConfigSet('rtwdemo_cgv', configObj);
cgvObj = cgv.CGV('rtwdemo_cgv');
cgvObj.activateConfigSet(configObj.Name);
```

See Also

Topics

“About Model Configurations” (Simulink)
“Programmatic Code Generation Verification”

addAdditionalHeaderFile

Add header file to array of header files for code replacement table entry

Syntax

```
addAdditionalHeaderFile(hEntry,headerFile)
```

Description

`addAdditionalHeaderFile(hEntry,headerFile)` adds a specified additional header file to the array of additional header files for a code replacement table entry.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Additional Header and Source Files

This example shows how to use the `addAdditionalHeaderFile` function with `addAdditionalIncludePath`, `addAdditionalSourceFile`, and `addAdditionalSourcePath` to specify additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
```

```
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

`hEntry` is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`headerFile` — Name of additional header file

character vector | string scalar

`headerFile` is a character vector or string scalar that specifies an additional header file.

Example: 'all_additions.h'

See Also

`addAdditionalIncludePath` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”

Introduced in R2007b

addAdditionalIncludePath

Add include path to array of include paths for code replacement table entry

Syntax

```
addAdditionalIncludePath(hEntry,path)
```

Description

`addAdditionalIncludePath(hEntry,path)` adds a specified additional include path to the array of additional include paths for a code replacement table entry.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Path to Additional Header and Source Files

This example shows how to use the `addAdditionalIncludePath` function with `addAdditionalHeaderFile`, `addAdditionalSourceFile`, and `addAdditionalSourcePath` to specify the path to additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
```

```
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* = `RTW.TflCFunctionEntry` or *`hEntry`* = `RTW.TflCOperationEntry`.

Example: `op_entry`

`path` — Path to an additional header file

character vector | string scalar

The *`path`* is a character vector or string scalar that specifies the full path to an additional header file. The character vector or string scalar can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB® workspace).

Example: `fullfile(libdir,'include')`

See Also

`addAdditionalHeaderFile` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”

Introduced in R2007b

addAdditionalLinkObj

Add link object to array of link objects for code replacement table entry

Syntax

```
addAdditionalLinkObj(hEntry,linkObj)
```

Description

`addAdditionalLinkObj(hEntry,linkObj)` adds a specified additional link object to the array of additional link objects for a code replacement table entry.

Examples

Specify an Additional Link Object

This example shows how to use the `addAdditionalLinkObj` function with `addAdditionalLinkObjPath` to specify an additional link object file fully for a code replacement table entry.

```
% Path to external object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
...
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *hEntry* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`.

Example: `op_entry`

`linkObj` — Name of an additional link object
character vector | string scalar

The *linkObj* is a character vector or string scalar that specifies an additional link object.

Example: `'addition.o'`

See Also

`addAdditionalLinkObjPath`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”

Introduced in R2007b

addAdditionalLinkObjPath

Add link object path to array of link object paths for code replacement table entry

Syntax

```
addAdditionalLinkObjPath(hEntry,path)
```

Description

`addAdditionalLinkObjPath(hEntry,path)` adds a specified additional link object path to the array of additional link object paths for a code replacement table entry.

Examples

Specify Path to Additional Link Object

This example shows how to use the `addAdditionalLinkObjPath` function with `addAdditionalLinkObj` to specify the path to an additional link object file fully for a code replacement table entry.

```
% Path to external object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
...
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *hEntry* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`.

Example: `op_entry`

path — Path to an additional link object
character vector | string scalar

The *path* is a character vector or string scalar that specifies the full path to an additional link object. The character vector or string scalar can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB workspace).

Example: `op_entry`

See Also

`addAdditionalLinkObj`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”

Introduced in R2008a

addAdditionalSourceFile

Add source file to array of source files for code replacement table entry

Syntax

```
addAdditionalSourceFile(hEntry,sourceFile)
```

Description

`addAdditionalSourceFile(hEntry,sourceFile)` adds a specified additional source file to the array of additional source files for a code replacement table entry.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Additional Header and Source Files

This example shows how to use the `addAdditionalSourceFile` function with `addAdditionalHeaderFile`, `addAdditionalIncludePath`, and `addAdditionalSourcePath` to specify additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
```

```
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* = `RTW.TflCFunctionEntry` or *`hEntry`* = `RTW.TflCOperationEntry`.

Example: `op_entry`

`sourceFile` — Name of an additional source file

character vector | string scalar

The *`sourceFile`* is a character vector or string scalar specifying an additional source file.

Example: `'all_additions.c'`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalSourcePath`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”

Introduced in R2007b

addAdditionalSourcePath

Add source path to array of source paths for code replacement table entry

Syntax

```
addAdditionalSourcePath(hEntry,path)
```

Description

`addAdditionalSourcePath(hEntry,path)` adds a specified additional source file path to the array of additional source file paths for a code replacement table.

This function adds `-I` to the compile line in the generated makefile.

Examples

Specify Path to Additional Header and Source Files

This example shows how to use the `addAdditionalSourcePath` function with `addAdditionalHeaderFile`, `addAdditionalIncludePath`, and `addAdditionalSourceFile` to specify path to additional header and source files fully for a code replacement table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
```

```
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* = `RTW.TflCFunctionEntry` or *`hEntry`* = `RTW.TflCOperationEntry`.

Example: `op_entry`

`path` — Path to an additional source file

character vector | string scalar

The *`path`* is a character vector or string scalar specifying the full path to an additional source file. The character vector or string scalar can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a character vector, cell array of character vectors, or string array in the MATLAB workspace).

Example: `fullfile(libdir, 'src')`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalSourceFile`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”

Introduced in R2007b

addAlgorithmProperty

Add algorithm properties for code replacement table entry

Syntax

`addAlgorithmProperty(`*`hEntry`*`, `*`name-value`*`)`

Arguments

`hEntry`

Handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

`name-value`

Algorithm property, specified as a comma-separated pair consisting of the name of an algorithm property and one or more algorithm values. Specify multiple values as a cell array of character vectors.

Name	Values
`'BPPower2Spacing'`	`'off'|'on'`
`'ExtrapMethod'`	`'Clip'|'Linear'`
`'IndexSearchMethod'`	`'Evenly spaced points'|'Linear search'|'Binary search'`
`'InterpMethod'`	`'Linear point-slope'|'Linear Lagrange'|'Flat'|'Nearest'`
`'RemoveProtectionInput'`	`'off'|'on'`
`'RndMeth'`	`'Ceiling'|'Convergent'|'Floor'|'Nearest'|'Round'|'Simplest'|'Zero'`
`'SaturateOnIntegerOverflow'`	`'off'|'on'`

Name	Values
'SupportTunableTableSize'	'off' | 'on'
'UseLastTableValue'	'off' | 'on'
'UseRowMajorAlgorithm'	'off' | 'on'

Description

The `addAlgorithmProperty` function adds algorithm property settings to the conceptual representation of a code replacement table entry. For example, use this function to adjust the algorithms applied by lookup table functions.

Examples

In the following example, the `addAlgorithmProperty` function configures the code generator to apply the following methods when replacing code for the `lookup1D` function:

- Clip extrapolation
- Linear interpolation
- Binary or linear index search

```
hLib = RTW.TflTable;

hEnt = RTW.TflCFunctionEntry;
hEnt.setTflCFunctionEntryParameters( ...
                    'Key',                      'lookup1D', ...
                    'Priority',                 100, ...
                    'ImplementationName',       'my_Lookup1D_Repl', ...
                    'ImplementationHeaderFile', 'my_Lookup1D.h', ...
                    'ImplementationSourceFile', 'my_Lookup1D.c', ...
                    'GenCallback', 'RTW.copyFileToBuildDir');

arg = hEnt.getTflArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);

arg = hEnt.getTflArgFromString('u1','double');
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u2','RTW_IO_INPUT','double');
```

```
arg.DimRange = [0 0; Inf Inf];
hEnt.addConceptualArg(arg);

arg = RTW.TflArgMatrix('u3', 'RTW_IO_INPUT', 'double');
arg.DimRange = [0 0; Inf Inf];
hEnt.addConceptualArg(arg);

hEnt.addAlgorithmProperty('ExtrapMethod', 'Clip');
hEnt.addAlgorithmProperty('InterpMethod', 'Linear point-slope');
hEnt.addAlgorithmProperty('IndexSearchMethod', 'Linear search');
```

See Also

`getTflArgFromString`

Topics

“Lookup Table Function Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2014b

addArgConf

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

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

Description

`addArgConf(`*`obj`*`, `*`portName`*`, `*`category`*`, `*`argName`*`, `*`qualifier`*`)` method 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 determines the argument position for the port in the function prototype, unless you change the order by other means, such as the `RTW.ModelSpecificCPrototype.setArgPosition` method.

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

Input Arguments

`obj`	Handle to a model-specific C prototype function control object previously returned by *`obj`* `= RTW.ModelSpecificCPrototype` or *`obj`* `= RTW.getFunctionSpecification(`*`modelName`*`)`.
`portName`	Character vector specifying the unqualified name of an inport or outport in your Simulink model.

category	Character vector specifying the argument category, either `'Value'` or `'Pointer'`.
argName	Character vector specifying a valid C identifier.
qualifier	Character vector specifying the argument type qualifier: `'none'`, `'const'`, `'const *'`, or `'const * const'`.

Examples

In the following example, you use the `addArgConf` method to add argument configuration information for ports `Input` and `Output` in an ERT-based version of `rtwdemo_counter`. After executing these commands, click the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to open the Model Interface 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)
```

Alternatives

You can use the Configure C Step Function Interface dialog box to customize the base rate C step function for a rate-based model. See “Override Default C Step Function Interface”.

See Also

`RTW.ModelSpecificCPrototype.attachToModel`

Topics

“Customize Generated C Function Interfaces”

addBaseline

Class: cgv.CGV
Package: cgv

Add baseline file for comparison

Syntax

```
cgvObj.addBaseline(inputName,baselineFile)
cgvObj.addBaseline(inputName,baselineFile,toleranceFile)
```

Description

`cgvObj.addBaseline(inputName,baselineFile)` associates a baseline data file to an `inputName` in *`cgvObj`*. *`cgvObj`* is a handle to a `cgv.CGV` object. If a baseline file is present, when you call `cgv.CGV.run`, *`cgvObj`* automatically compares baseline data to the result data of the current execution of *`cgvObj`*.

`cgvObj.addBaseline(inputName,baselineFile,toleranceFile)` includes an optional tolerance file to apply when comparing the baseline data to the result data of the current execution of *`cgvObj`*.

Input Arguments

`inputName`

A unique numeric or character identifier assigned to the input data associated with `baselineFile`

`baselineFile`

A MAT-file containing baseline data

`toleranceFile`

File containing the tolerance specification, which is created using `cgv.CGV.createToleranceFile`

Examples

A typical workflow for defining baseline data in a `cgv.CGV` object and then comparing the baseline data to the execution data is as follows:

1. Create a `cgv.CGV` object for a model.
2. Add input data to the `cgv.CGV` object by calling `cgv.CGV.addInputData`.
3. Add the baseline file to the `cgv.CGV` object by calling `cgv.CGV.addBaseline`. which associates the `inputName` for input data in the `cgv.CGV` object with input data stored in the `cgv.CGV` object as the baseline data.
4. Run the `cgv.CGV` object by calling `cgv.CGV.run`, which automatically compares the baseline data to the result data in this execution.
5. Call `cgv.CGV.getStatus` to determine the results of the comparison.

See Also

`cgv.CGV.addInputData` | `cgv.CGV.createToleranceFile` | `cgv.CGV.getStatus` | `cgv.CGV.run`

Topics

“Verify Numerical Equivalence with CGV”

addHeaderReportFcn

Class: cgv.CGV
Package: cgv

Add callback function to execute before executing input data in object

Syntax

cgvObj.addHeaderReportFcn(CallbackFcn)

Description

cgvObj.addHeaderReportFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* before executing input data included in *cgvObj*. The callback function signature is:

```
CallbackFcn(cgvObj)
```

Examples

The callback function, *HeaderReportFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addHeaderReportFcn(@HeaderReportFcn);
```

where *HeaderReportFcn* is defined as:

```
function HeaderReportFcn(cgvObj)
...
end
```

See Also

cgv.CGV.run

Topics

“Callbacks for Customized Model Behavior” (Simulink)

addPostExecFcn

Class: cgv.CGV
Package: cgv

Add callback function to execute after each input data file is executes

Syntax

cgvObj.addPostExecFcn(CallbackFcn)

Description

cgvObj.addPostExecFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* after each input data file is executed for the model. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numerical identifier associated with input data in the *cgvObj*.

Examples

The callback function, *PostExecutionFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addPostExecFcn(@PostExecutionFcn);
```

where *PostExecutionFcn* is defined as:

```
function PostExecutionFcn(cgvObj, inputIndex)
...
end
```

See Also

cgv.CGV.run

Topics

“Callbacks for Customized Model Behavior” (Simulink)

addPostExecReportFcn

Class: cgv.CGV
Package: cgv

Add callback function to execute after each input data file executes

Syntax

cgvObj.addPostExecReportFcn(CallbackFcn)

Description

cgvObj.addPostExecReportFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* after each input data file is executed for the model. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numeric identifier associated with input data in the *cgvObj*.

Examples

The callback function, *PostExecutionReportFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addPostExecReportFcn(@PostExecutionReportFcn);
```

where *PostExecutionReportFcn* is defined as:

```
function PostExecutionReportFcn(cgvObj, inputIndex)
...
end
```

See Also

cgv.CGV.run

Topics

“Callbacks for Customized Model Behavior” (Simulink)

addPreExecFcn

Class: cgv.CGV
Package: cgv

Add callback function to execute before each input data file executes

Syntax

cgvObj.addPreExecFcn(CallbackFcn)

Description

cgvObj.addPreExecFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run calls *CallbackFcn* before executing each input data file in *cgvObj*. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numeric identifier associated with input data in *cgvObj*.

Examples

The callback function, *PreExecutionFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addPreExecFcn(@PreExecutionFcn);
```

where *PreExecutionFcn* is defined as:

```
function PreExecutionFcn(cgvObj, inputIndex)
...
end
```

See Also

cgv.CGV.run

Topics

“Callbacks for Customized Model Behavior” (Simulink)

addPreExecReportFcn

Class: cgv.CGV
Package: cgv

Add callback function to execute before each input data file executes

Syntax

`cgvObj.addPreExecReportFcn(CallbackFcn)`

Description

`cgvObj.addPreExecReportFcn(CallbackFcn)` adds a callback function to *`cgvObj`*. *`cgvObj`* is a handle to a `cgv.CGV` object. `run` calls *`CallbackFcn`* before executing each input data file in *`cgvObj`*. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

`inputIndex` is a unique numerical identifier associated with input data in *`cgvObj`*.

Examples

The callback function, *`PreExecutionReportFcn`*, is added to `cgv.CGV` object, *`cgvObj`*

```
cgvObj.addPreExecReportFcn(@PreExecutionReportFcn);
```

where *`PreExecutionReportFcn`* is defined as:

```
function PreExecutionReportFcn(cgvObj, inputIndex)
...
end
```

See Also

`cgv.CGV.run`

Topics

“Callbacks for Customized Model Behavior” (Simulink)

addTrailerReportFcn

Class: cgv.CGV
Package: cgv

Add callback function to execute after the input data executes

Syntax

cgvObj.addTrailerReportFcn(CallbackFcn)

Description

cgvObj.addTrailerReportFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. run executes the input data files in *cgvObj* and then calls *CallbackFcn*. The callback function signature is:

```
CallbackFcn(cgvObj)
```

Examples

The callback function, *TrailerReportFcn*, is added to cgv.CGV object, *cgvObj*

```
cgvObj.addTrailerReportFcn(@TrailerReportFcn);
```

where *TrailerReportFcn* is defined as:

```
function TrailerReportFcn(cgvObj)
...
end
```

See Also

cgv.CGV.run

Topics

“Callbacks for Customized Model Behavior” (Simulink)

addCheck

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Add checks

Syntax

`addCheck(obj, checkID)`

Description

`addCheck(obj, checkID)` includes the check, *checkID*, in the Code Generation Advisor. When a user selects the objective, the Code Generation Advisor includes the check, unless another objective with a higher priority excludes the check.

Input Arguments

obj	Handle to a code generation objective object previously created.
checkID	Unique identifier of the check that you add to the new objective.

Examples

Add the **Identify questionable code instrumentation (data I/O)** check to the objective.

```
addCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

See Also

`Simulink.ModelAdvisor`

Topics

“Create Custom Code Generation Objectives”
`Simulink.ModelAdvisor`

addComplexTypeAlignment

Specify alignment boundary of a complex type

Syntax

```
addComplexTypeAlignment(hDataAlign,baseType,alignment)
```

Description

`addComplexTypeAlignment(hDataAlign,baseType,alignment)` specifies the alignment boundary of real and complex data members of a complex type.

The starting memory address of the real and imaginary part of complex variables produced by the code generator with the specified type are a multiple of the specified alignment boundary. The code generator replaces operations in generated code when both of these conditions are true:

- A code replacement table entry has a complex argument with a data alignment requirement that is less than or equal to the alignment boundary value
- The entry satisfies all other code replacement match criteria.

To use this function, your code replacement library registration file must include additional compiler data alignment information, such as alignment syntax.

Examples

Specify Alignment Boundary for Complex Types

This example shows how to specify a 16-byte alignment boundary for complex `int8` types by adding the `addComplexTypeAlignment` line to your code replacement library registration file.

```
function rtwTargetInfo(cm)
% rtwTargetInfo function to register a code replacement library (CRL)
```

```
% for use with  code generation

  % Register the CRL defined in local function locCrlRegFcn
  cm.registerTargetInfo(@locCrlRegFcn);

end % End of RTWTARGETINFO

% Local function to define a CRL containing crl_table_mmul_4x4_single_align
function thisCrl = locCrlRegFcn

  % create an alignment specification object, assume gcc
  as = RTW.AlignmentSpecification;
  as.AlignmentType = {'DATA_ALIGNMENT_LOCAL_VAR', ...
                      'DATA_ALIGNMENT_GLOBAL_VAR', ...
                      'DATA_ALIGNMENT_STRUCT_FIELD'};
  as.AlignmentSyntaxTemplate = '__attribute__((aligned(%n)))';
  as.SupportedLanguages={'c', 'c++'};

  % add the alignment specification object
  da = RTW.DataAlignment;
  da.addAlignmentSpecification(as);
  da.addComplexTypeAlignment('int8', 16);

  % add the data alignment object to target characteristics
  tc = RTW.TargetCharacteristics;
  tc.DataAlignment = da;

  % Instantiate a CRL registry entry
  thisCrl = RTW.TflRegistry;

  % Define the CRL properties
  thisCrl.Name = 'Data Alignment Example';
  thisCrl.Description = 'Example of replacement with data alignment';
  thisCrl.TableList = {'crl_table_mmul_4x4_single_align'};
    thisCrl.TargetCharacteristics = tc;

end % End of LOCCRLREGFCN
```

Input Arguments

`hDataAlign` — Handle to a data alignment object

handle

The *`hDataAlign`* is a handle to a data alignment object, previously returned by *`hDataAlign`* `= RTW.DataAlignment`.

Example: `da`

`baseType` — Specifies a built-in data type

character vector | string scalar

The *`baseType`* is a character vector or string scalar that specifies a built-in data type such as `int8` or `long`.

Example: `'int8'`

`alignment` — Specifies the alignment boundary
positive integer

The *`alignment`* is a positive integer that is a power of 2 and does not exceed 128. This value specifies the alignment boundary.

Example: 16

See Also

Topics

“Data Alignment for Code Replacement”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2014a

addConceptualArg

Add conceptual argument to array of conceptual arguments for code replacement table entry

Syntax

`addConceptualArg(hEntry,arg)`

Description

`addConceptualArg(hEntry,arg)` adds a specified conceptual argument to the array of conceptual arguments for a code replacement table entry.

Examples

Add Conceptual Arguments for Ports

This example shows how the `addConceptualArg` function adds conceptual arguments for the output operand and the two input operands for an addition operation.

```
hLib = RTW.TflTable;

% Create entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
```

```
op_entry.addConceptualArg(arg);

arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg(arg);

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg(arg);

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *`hEntry`* `=` `RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

`arg` — Argument added to the array of conceptual arguments

character vector | string scalar

The *`arg`* is the argument, such as returned by *`arg`* `= getTflArgFromString(`*`name`*`,` *`datatype`*`)`, added to the array of conceptual arguments for the code replacement table entry.

Example: `'hLib.getTflArgFromString('y1','uint8')'`

See Also

`getTflArgFromString`

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

addDWorkArg

Add DWork argument for semaphore entry in code replacement table

Syntax

`addDWorkArg(hEntry,arg)`

Description

`addDWorkArg(hEntry,arg)` adds a specified DWork argument to the arguments for a semaphore entry in a code replacement table.

Examples

Add a DWork Argument

This example shows how to use the `addDWorkArg` function to add a DWork argument named `d1` to the arguments for a semaphore entry in a code replacement table.

```
hLib = RTW.TflTable;

% specify semaphore init function.
hEnt = RTW.TflCSemaphoreEntry;
hEnt.setTflCSemaphoreEntryParameters( ...
  'Key',                      'RTW_SEM_INIT', ...
  'Priority',                 30, ...
  'ImplementationName',       'mySemCreate', ...
  'ImplementationHeaderFile', 'mySem.h', ...
  'ImplementationSourceFile', 'mySem.c', ...
  'ImplementationHeaderPath', LibPath, ...
  'ImplementationSourcePath', LibPath, ...
  'GenCallback',              'RTW.copyFileToBuildDir', ...
  'SideEffects',              true);

% specify conceptual operands and result
```

```
arg = hLib.getTflArgFromString('y1', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);
arg = hLib.getTflArgFromString('u1', 'void');
hEnt.addConceptualArg(arg);

% specify replacement function signature
arg=hLib.getTflArgFromString('y1','void');
hEnt.Implementation.setReturn(arg);
arg.IOType = 'RTW_IO_OUTPUT';

% DWork Arg
arg = hLib.getTflDWorkFromString('d1','void*');
hEnt.addDWorkArg(arg);

addEntry(hLib, hEnt);
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement semaphore table entry class, using *`hEntry`* = `RTW.TflCSemaphoreEntry`.

Example: `sem_entry`

`arg` — Argument added to the arguments for the table entry

character vector | string scalar

Argument, such as returned by *`arg`* = `getTflDWorkFromString(`*`name`*`, `*`datatype`*`)`, added to the arguments for the code replacement table entry.

Example: `'hLib.getTflDWorkFromString('d1','void*')'`

See Also

`getTflDWorkFromString`

Topics

“Semaphore and Mutex Function Replacement”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2011b

addConfigSet

Class: cgv.CGV
Package: cgv

Add configuration set

Syntax

```
cgvObj.addConfigSet(configSet)
cgvObj.addConfigSet('configSetName')
cgvObj.addConfigSet('file','configSetFileName')
cgvObj.addConfigSet('file','configSetFileName','variable',
'configSetName')
```

Description

`cgvObj.addConfigSet(configSet)` is an optional method that adds the configuration set to the object. *`cgvObj`* is a handle to a `cgv.CGV` object. *`configSet`* is a variable that specifies a configuration set.

`cgvObj.addConfigSet('configSetName')` is an optional method that adds the configuration set to the object. *`configSetName`* is a character vector that specifies the name of the configuration set in the workspace.

`cgvObj.addConfigSet('file','configSetFileName')` is an optional method that adds the configuration set to the object. *`configSetFileName`* is a character vector that specifies the name of the file that contains only one configuration set.

`cgvObj.addConfigSet('file','configSetFileName','variable',
'configSetName')` is an optional method that adds the configuration set to the object. The file contains one or more configuration sets. Specify the name of the configuration set to use.

This method replaces the configuration parameter values in the model with the values from the configuration set that you add. The object applies the configuration set when you call the `run` method. You can add only one configuration set for each `cgv.CGV` object.

See Also

Topics

“About Model Configurations” (Simulink)
“Programmatic Code Generation Verification”

addEntry

Add table entry to collection of table entries registered in code replacement table

Syntax

`addEntry(hTable,entry)`

Description

`addEntry(hTable,entry)` adds a function or operator entry that you have constructed to the collection of table entries registered in a code replacement table.

Examples

Add Operator Entry to Code Replacement Table

This example shows how to use the `addEntry` function to add an operator entry to a code replacement table after the entry is constructed.

```
hLib = RTW.TflTable;

% Create an entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg( arg );
```

```
arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg( arg );

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

Input Arguments

`hTable` — Handle to a code replacement table

handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* = `RTW.TflTable`.

Example: `hLib`

`entry` — Handle to a function or operator entry

handle

The *`entry`* is a handle to a function or operator entry that you have constructed after calling *`hEntry`* = `RTW.TflCFunctionEntry` or *`hEntry`* = `RTW.TflCOperationEntry`.

Example: `op_entry`

See Also

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

addInputData

Class: cgv.CGV
Package: cgv

Add input data

Syntax

```
cgvObj.addInputData(inputName, inputDataFile)
```

Description

`cgvObj.addInputData(inputName, inputDataFile)` adds an input data file to *`cgvObj`*. *`cgvObj`* is a handle to a `cgv.CGV` object. `inputName` is a unique identifier, which *`cgvObj`* associates with the input data in `inputDataFile`.

Input Arguments

`inputName`

`inputName` is a unique numeric or character identifier, which is associated with the input data in `inputDataFile`.

`inputDataFile`

`inputDataFile` is an input data file, with or without the `.mat` extension. *`cgvObj`* uses the input data when the model executes during `cgv.CGV.run`. If the input file is in the working folder, the *`cgvObj`* does not require the path. `addInputData` does not qualify that the contents of `inputDataFile` relate to the inputs of the model. Data that is not used by the model will not throw a warning or error.

Tips

- When calling `addInputData` you can modify configuration parameters by including their settings in the input file, `inputDataFile`.
- If you omit calling `addInputData` before executing the model, the `cgv.CGV` object runs once using data in the base workspace.
- The *`cgvObj`* uses the `inputName` to identify the input data associated with output data and output data files. *`cgvObj`* passes `inputName` to a callback function to identify the input data that the callback function uses.

See Also

`cgv.CGV.run`

Topics

“Verify Numerical Equivalence with CGV”

addParam

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Add parameters

Syntax

`addParam(obj, paramName, value)`

Description

`addParam(obj, paramName, value)` adds a parameter to the objective, and defines the value of the parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**.

Input Arguments

obj	Handle to a code generation objective object previously created.
paramName	Parameter that you add to the objective.
value	Value of the parameter.

Examples

Add `DefaultParameterBehavior` to the objective, and specify the parameter value as `Inlined`.

```
addParam(obj, 'DefaultParameterBehavior', 'Inlined');
```

See Also

`get_param`

Topics

“Create Custom Code Generation Objectives”

addPostLoadFiles

Class: cgv.CGV
Package: cgv

Add files required by model

Syntax

`cgvObj.addPostLoadfiles({FileList})`

Description

`cgvObj.addPostLoadfiles({FileList})` is an optional method that adds a list of MATLAB and MAT-files to the object. *`cgvObj`* is a handle to a `cgv.CGV` object. *`cgvObj`* executes and loads the files after opening the model and before running tests. *`FileList`* is a cell array of names of MATLAB and MAT-files in the testing directory that the model requires to run.

Note Subsequent *`cgvObj`*`.addPostLoadFiles` calls to the same `cgv.CGV` object replaces the list of MATLAB and MAT-files of that object.

See Also

Topics

“Verify Numerical Equivalence with CGV”
“Callbacks for Customized Model Behavior” (Simulink)

annotate

Color profiled model components or open model with profiled components colored

Syntax

`annotate(`*`executionProfile`*`)`

Description

When you run a SIL or PIL simulation with code execution profiling, the software generates the workspace variable *`executionProfile`*, specified in **Configuration Parameters > Code Generation > Verification > Workspace variable**.

`annotate(`*`executionProfile`*`)` colors the profiled model components blue. If the model is closed, this command opens the model, with profiled components colored blue. Clicking a blue component opens a window that displays execution-time metrics for generated code.

See Also

`report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

Introduced in R2016b

attachToModel

Class: RTW.ModelCPPClass
Package: RTW

Attach model-specific C++ class interface to loaded ERT-based Simulink model

Syntax

`attachToModel(obj, modelName)`

Description

`attachToModel(obj, modelName)` attaches a model-specific C++ class interface to a loaded ERT-based Simulink model.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass on page 1-412` or *obj* = `RTW.ModelCPPDefaultClass on page 1-418`.
modelName	Character vector specifying the name of a loaded ERT-based Simulink model to which the object is going to be attached.

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C ++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

attachToModel

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

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

Description

`attachToModel(`*`obj`*`, `*`modelName`*`)` attaches a model-specific C function prototype to a loaded ERT-based Simulink model.

Input Arguments

`obj`	Handle to a model-specific C prototype function control object previously returned by *`obj`* `= RTW.ModelSpecificCPrototype`.
`modelName`	Character vector specifying the name of a loaded ERT-based Simulink model to which the object is going to be attached.

Alternatives

Use the Configure C Step Function Interface dialog box to customize the base rate C step function for a rate-based model. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

cgv.CGV class

Package: cgv

Verify numerical equivalence of results

Description

Executes a model in different environments such as, simulation, Software-In-the-Loop (SIL), or Processor-In-the-Loop (PIL) and stores numerical results. Using the `cgv.CGV` class methods, you can create a script to verify that the model and the generated code produce numerically equivalent results.

`cgv.CGV` and `cgv.Config` use two of the same properties. Before executing a `cgv.CGV` object, use `cgv.Config` to verify the model configured for the mode of execution that you specify. If the top model is set to normal simulation mode, referenced models set to PIL mode are changed to Accelerator mode.

Construction

`cgvObj = cgv.CGV(model_name)` creates a handle to a code generation verification object using the default parameter values. `model_name` is the name of the model that you are verifying.

`cgvObj = cgv.CGV(model_name,Name,Value)` constructs the object using the parameter values, specified as `Name,Value` pair arguments. Parameter names and values are not case sensitive.

Input Arguments

`model_name`

Name of the model that you are verifying.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`).

You can specify several name-value pair arguments in a variety of orders, such as `Name1,Value1,…,NameN,ValueN`.

`ComponentType`

Define the SIL or PIL approach

Value	Description
`topmodel` (default)	Top-model SIL or PIL simulation and standalone code interface mode.
`modelblock`	Model block SIL or PIL simulation and model reference target code interface mode.

If mode of execution is simulation (`Connectivity` is `sim`), choosing either value for `ComponentType` does not alter simulation results.

Default: topmodel

`Connectivity`

Specify mode of execution

Value	Description
`sim` or `normal` (default)	Mode of execution is Normal simulation.
`sil`	Mode of execution is SIL.
`pil`	Mode of execution is PIL.

Properties

`Description`

Specify a description of the object.

Default: `' '` (null character vector)

`Name`

Specify a name for the object.

Default: `' '` (null character vector)

Methods

activateConfigSet	Activate configuration set of model
addBaseline	Add baseline file for comparison
addHeaderReportFcn	Add callback function to execute before executing input data in object
addPostExecFcn	Add callback function to execute after each input data file is executes
addPostExecReportFcn	Add callback function to execute after each input data file executes
addPreExecFcn	Add callback function to execute before each input data file executes
addPreExecReportFcn	Add callback function to execute before each input data file executes
addTrailerReportFcn	Add callback function to execute after the input data executes
addConfigSet	Add configuration set
addInputData	Add input data
addPostLoadFiles	Add files required by model
compare	Compare signal data
copySetup	Create copy of `cgv.CGV` object
createToleranceFile	Create file correlating tolerance information with signal names
getOutputData	Get output data
getSavedSignals	Display list of signal names to command line
getStatus	Return execution status
plot	Create plot for signal or multiple signals
run	Execute CGV object
setMode	Specify mode of execution
setOutputDir	Specify folder
setOutputFile	Specify output data file name

Copy Semantics

Handle. To learn how handle classes change copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

Examples

The general workflow for testing a model for numerical equivalence using the `cgv.CGV` class is to:

1. Create a `cgv.CGV` object, `cgvObj`, for each mode of execution and use the `cgv.CGV` set up methods to configure the model for each execution. The set up methods are:
   - `addInputData`
   - `addPostLoadFiles`
   - `setOutputDir`
   - `setOutputFile`
   - `addCallBack`
   - `addConfigSet`
2. Run the model for each mode of execution using the `cgvObj.run` method.
3. Use the `cgv.CGV` access methods to get and evaluate the data. The access methods are:
   - `getOutputData`
   - `getSavedSignals`
   - `plot`
   - `compare`

An object should be run only once. After the object is run, the set up methods are not used for that object. You then use the access methods for verifying the numerical equivalence of the results.

Note Simulink Test™ is a separate product that provides additional capabilities for SIL and PIL testing, for example, test sequence construction and test management.

See Also

`cgv.Config`

Topics

“Verify Numerical Equivalence with CGV”
“Using Code Generation Verification API”

cgv.Config class

Package: cgv

Check and modify model configuration parameter values

Description

Creates a handle to a `cgv.Config` object that supports checking and optionally modifying models for compatibility with various modes of execution that use generated code, such as, Software-In-the-Loop (SIL) or Processor-In-the-Loop (PIL).

To execute the model in the mode that you specify, you might need to make additional modifications to the configuration parameter values or the model beyond those configured by the `cgv.Config` object.

By default, `cgv.Config` modifies configuration parameter values to the values that it recommends, but does not save the model. Alternatively, you can use `cgv.Config` parameters to modify the default specification. For more information, see the properties, `ReportOnly` and `SaveModel`.

If you use `cgv.Config` to modify a model, do not use referenced configuration sets in that model. If a model uses a referenced configuration set, update the model with a copy of the configuration set, by using the `Simulink.ConfigSetRef.getRefConfigSet` method.

If you use `cgv.Config` on a model that executes a callback function, the callback function might modify configuration parameter values each time the model loads. The callback function might revert changes that `cgv.Config` made. If this change occurs, the model might not be set up for SIL or PIL. For more information, see “Callbacks for Customized Model Behavior” (Simulink).

Construction

`cfgObj = cgv.Config(model_name)` creates a handle to a `cgv.Config` object, `cfgObj`, using default values for properties. `model_name` is the name of the model that you are checking and optionally configuring.

`cfgObj = cgv.Config(model_name, Name, Value)` constructs the object using options, specified as parameter name and value pairs. Parameter names and values are not case sensitive.

`Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in a variety of orders, such as `Name1,Value1,…,NameN,ValueN`.

Properties

`CheckOutports`

Specify whether to compile the model and check that the model outports configuration is compatible with the `cgv.CGV` object. If your script fixes errors reported by `cgv.Config`, you can set `CheckOutports` to `off`.

Value	Description
`on` (default)	Compile the model and check the model outports configuration
`off`	Do not compile the model or check the model outports configuration

`ComponentType`

Define the SIL or PIL approach

If mode of execution is simulation (`connectivity` is `sim`), choosing either value for `ComponentType` does not alter simulation results. However, `cgv.Config` recommends configuration parameter values based on the value of `ComponentType`.

Value	Description
`topmodel` (default)	Top-model SIL or PIL simulation and standalone code interface mode.
`modelblock`	Model block SIL or PIL simulation and model reference target code interface mode.

`Connectivity`

Specify mode of execution

Value	Description
`sim` (default)	Mode of execution is simulation. Recommends changes to a subset of the configuration parameters that SIL and PIL targets require.
`sil`	Mode of execution is SIL. Requires that the system target file is set to `'ert.tlc'` and that you do not use your own external target. Recommends changes to the configuration parameters that SIL targets require.
`pil`	Mode of execution is PIL with custom connectivity that you provide using the PIL Connectivity API. Recommends changes to the configuration parameters that PIL targets with custom connectivity require.

`LogMode`

Specify the **Signal logging** and **Output** parameters on the **Data Import/Export** pane of the Configuration Parameters dialog box.

Value	Description
`SignalLogging`	Log signal data to a MATLAB workspace variable during execution. This parameter selects the **Data Import/ Export > Signal logging** parameter in the Configuration Parameters dialog box.

Value	Description
`SaveOutput`	Save output data to a MATLAB workspace variable during execution. This parameter selects **Data Import/ Export** > **Output** parameter in the Configuration Parameters dialog box. The **Output** parameter does not save bus outputs.

`ReportOnly`

The `ReportOnly` property specifies whether `cgv.Config` modifies the recommended values of the configuration parameters of the model.

If you set `ReportOnly` to `on`, `SaveModel` must be `off`.

Value	Description
`off` (default)	`cgv.Config` automatically modifies the configuration parameter values that it recommends for the model.
`on`	`cgv.Config` does not modify the configuration parameter values that it recommends for the model.

`SaveModel`

Specify whether to save the model with the configuration parameter values recommended by `cgv.Config`.

If you set `SaveModel` to `'on'`, `ReportOnly` must be `'off'`.

Value	Description
`off` (default)	Do not save the model.
`on`	Save the model in the working folder.

Methods

configModel	Determine and change configuration parameter values
displayReport	Display results of comparing configuration parameter values
getReportData	Return results of comparing configuration parameter values

Copy Semantics

Handle. To learn how handle classes change copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

Examples

Configure the `rtwdemo_iec61508` model for top-model SIL. Then view the changes at the MATLAB Command Window:

```
% Create a cgv.Config object and configure the model for top-model SIL.
load_system('rtwdemo_iec61508');
set_param('rtwdemo_iec61508', 'SaveFormat', 'StructureWithTime');
cgvCfg = cgv.Config('rtwdemo_iec61508', 'LogMode', 'SaveOutput', ...
    'connectivity', 'sil');
cgvCfg.configModel();

% Display the results of what the cgv.Config object changed.
cgvCfg.displayReport();

% Close the rtwdemo_iec61508 model.
bdclose('rtwdemo_iec61508');
```

See Also

`cgv.CGV`

Topics

“Programmatic Code Generation Verification”

coder.dataAlignment

Specify data alignment for global or entry-point/exported function input and output arguments

Syntax

```
coder.dataAlignment('varName',align_value)
```

Description

`coder.dataAlignment('varName',align_value)` specifies data alignment in MATLAB code for the variable (`varName`), which is imported data or global (exported) data. The code generator aligns the imported or exported data to the alignment boundary (`align_value`).

Examples

Data Alignment for Imported Data

An example function that specifies data alignment for imported data.

```
function y = importedDataExampleFun(x1,x2)

coder.dataAlignment('x1',16);      % Specifies information
coder.dataAlignment('x2',16);    % Specifies information
coder.dataAlignment('y',16);     % Specifies information

y = x1 + x2;

end
```

Data Alignment for Exported Data

An example function that specifies data alignment for exported data.

```
function a = exportedDataExampleFun(b)

global z;
coder.dataAlignment('z',8);

a = b + z;

end
```

Input Arguments

`'varName'` — Variable name

character array

The *`varName`* is a character array of the variable name that requires alignment information specification.

`align_value` — Data alignment boundary value

integer

The *`align_value`* is an integer number which should be a power of 2, from 2 through 128. This number specifies the power-of-2 byte alignment boundary.

Limitations

Limitations on variables supported by `coder.dataAlignment` directive:

- Only use `coder.dataAlignment` to specify alignment information for function inputs, outputs, and global variables.
- `coder.dataAlignment` supports only matrix types, including matrix of complex types.
- For exported custom storage classes (CSCs), `coder.dataAlignment` supports only `ExportedGlobal`. You can specify alignment information for any imported CSCs.
- The code generator ignores `coder.dataAlignment` for non-ERT or non-ERT derived system target files.

- Global variables tagged using the `coder.dataAlignment` directive from within a MATLAB function block are ignored. Set the alignment value on the corresponding Data Store Memory.

See Also

`codegen`

Topics

“Data Alignment for Code Replacement”
“Define Code Replacement Mappings”
“What Is Code Replacement Customization?”
“What Is Code Replacement?”

Introduced in R2017a

coder.replace

Replace current MATLAB function implementation with code replacement library function in generated code

Syntax

```
coder.replace(ifNoReplacement)
```

Description

`coder.replace(ifNoReplacement)` replaces the current function implementation with a code replacement library function.

During code generation, when you call `coder.replace` in a MATLAB function, the code generator performs a code replacement library lookup for the function signature:

```
[y1_type, y2_type,..., yn_type]=fcn(x1_type, x2_type,...,xn_type)
```

The input data types are `x1_type, x2_type,...,xn_type` and the output types, derived from the implementation, are `y1_type, y2_type,..., yn_type`. If a match for the MATLAB function is found in a registered code replacement library, the contents of the MATLAB function are discarded and replaced with a call to the code replacement library function. If a match is not found, the code generates without replacement.

`coder.replace` only affects code generation and does not alter MATLAB code or MEX function generation. `coder.replace` is intended to replace a MATLAB function that has behavior equivalent to its replacement function implementation. If the MATLAB function body is empty or not equivalent to the replacement function implementation, it may be eliminated from the generated code. The MATLAB function prior to replacement is used for simulation. You are responsible for verifying the numeric result of simulation and code generation after replacement.

Examples

Replace a MATLAB Function with Custom Code

Replace a MATLAB function with a custom implementation that is registered in the code replacement library.

Write a MATLAB function, `calculate`, that you want to replace with a custom implementation, `replacement_calculate_impl.c`, in the generated code.

```
function y = calculate(x)
% Search in the code replacement library for replacement
% and use replacement function if available
% Error if not found
  coder.replace('-errorifnoreplacement');
  y = sqrt(x);
end
```

Write a MATLAB function, `top_function`, that calls `calculate`.

```
function out = top_function(in)
  p = calculate(in);
  out = exp(p);
end
```

Create a file named `crl_table_calculate.m` that describes the function entries for a code replacement table. The replacement function `replacement_calculate_impl.c` and header file `replacement_calculate_impl.h` must be on the path.

```
hLib = RTW.TflTable;

%---------- entry: calculate -----------
hEnt = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(hEnt, ...
  'Key', 'calculate', ...
  'Priority', 100, ...
  'ArrayLayout', 'COLUMN_MAJOR', ...
  'ImplementationName', ...
  'replacement_calculate_impl', ...
  'ImplementationHeaderFile', ...
  'replacement_calculate_impl.h', ...
  'ImplementationSourceFile', ...
  'replacement_calculate_impl.c')
% Conceptual Args

arg = getTflArgFromString(hEnt, 'y1','double');
```

```
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);

arg = getTflArgFromString(hEnt, 'u1','double');
addConceptualArg(hEnt, arg);

% Implementation Args

arg = getTflArgFromString(hEnt, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.Implementation.setReturn(arg);

arg = getTflArgFromString(hEnt, 'u1','double');
hEnt.Implementation.addArgument(arg);

addEntry(hLib, hEnt);
```

Create an `rtwTargetInfo` file:

```
function rtwTargetInfo(tr)
% rtwTargetInfo function to register a code
% replacement library (CRL)
% for use with codegen

  % Register the CRL defined in local function locCrlRegFcn
  tr.registerTargetInfo(@locCrlRegFcn);

end % End of RTWTARGETINFO

function thisCrl = locCrlRegFcn

  % Instantiate a CRL registry entry
  thisCrl = RTW.TflRegistry;

  % Define the CRL properties
  thisCrl.Name = 'My calculate Example';
  thisCrl.Description = 'Demonstration of function replacement';
  thisCrl.TableList = {'crl_table_calculate'};
  thisCrl.BaseTfl = 'C89/C90 (ANSI)';
  thisCrl.TargetHWDeviceType = {'*'};

end % End of LOCCRLREGFCN
```

Refresh registration information. At the MATLAB command line, enter:

```
RTW.TargetRegistry.getInstance('reset');
```

Because the data type of `x` and `y` is `double`, `coder.replace` searches for `double = calculate(double)` in the Code Replacement Library. If it finds a match, `codegen` generates the following code:

```
real_T top_function(real_T in)
{
  real_T p;
  p = replacement_calculate_impl(in);
  return exp(p);
}
```

In the generated code, the replacement function `replacement_calculate_impl` replaces the MATLAB function `calculate`.

Input Arguments

`ifNoReplacement` — Selects whether the code generator produces a warning or error when no match is found

`'-errorifnoreplacement'` | `'-warnifnoreplacement'`

The *`ifNoReplacement`* argument selects whether the code generator issues an error or warning when no match is found. If this argument is omitted, the code generator does not issue an error or warning.

`coder.replace('-errorifnoreplacement')` replaces the current function implementation with a code replacement library function. If a match is not found, code generation stops. An error message describing the code replacement library lookup failure is generated.

`coder.replace('-warnifnoreplacement')` replaces the current function implementation with a code replacement library function. If match is not found, code is generated for the current function. A warning describing the code replacement library lookup failure is generated during code generation.

Example: `coder.replace()`

Tips

- `coder.replace` requires an Embedded Coder license.
- `coder.replace` is a code generation function and does not alter MATLAB code or MEX function generation.
- `coder.replace` is not intended to be called multiple times within a function.
- `coder.replace` is not intended to be used within conditional expressions and loops.
- `coder.replace` does not support saturation and rounding modes during code replacement library lookups.
- `coder.replace` does not support `varargout`.
- `coder.replace` does not support function replacement that requires data alignment.
- `coder.replace` does not support function replacement of MATLAB functions with variable-size inputs.

See Also

`codegen`

Topics

“Replace MATLAB Functions with Custom Code Using coder.replace”
“Replace MATLAB Functions Specified in MATLAB Function Blocks”
“Define Code Replacement Mappings”
“What Is Code Replacement Customization?”
“What Is Code Replacement?”

Introduced in R2012b

coder.setupMISRAConfig

Configure code generation parameters to increase code compliance with MISRA C:2012 guidelines

Syntax

```
coder.setupMISRAConfig(cfg)
```

Description

`coder.setupMISRAConfig(cfg)` sets up an Embedded Coder code generation configuration object to increase the likelihood of generating code that complies with MISRA C®:2012 guidelines.

Examples

Configure Code Generation Parameters for Increased MISRA C Compliance

Create an Embedded Coder code generation configuration object.

```
cfg = coder.config('lib', 'ecoder', true);
```

Set properties that might impact MISRA C compliance.

```
coder.setupMISRAConfig(cfg);
```

`coder.setupMISRAConfig` sets property values according to the values in this table.

Embedded Coder Configuration Object Property	Value for Increased MISRA C Compliance
`CastingMode`	`'Standards'`
`DataTypeReplacement`	`'CoderTypedefs'`

Embedded Coder Configuration Object Property	Value for Increased MISRA C Compliance
`DynamicMemoryAllocation`	`'Off'`
`EnableRuntimeRecursion`	`false`
`EnableSignedLeftShifts`	`false`
`EnableSignedRightShifts`	`false`
`GenerateDefaultInSwitch`	`true`
`ParenthesesLevel`	`'Maximum'`
`TargetLangStandard`	`'C99 (ISO)'` for C, `'C++03 (ISO)'` for C++

Input Arguments

`cfg` — Embedded Coder code generation configuration object

`coder.EmbeddedCodeConfig` object

Embedded Coder configuration object for generating C/C++ code from MATLAB code. Create the object by using `coder.config`.

Example: cfg = coder.config('lib', 'ecoder',true)

See Also

Topics

“Increase Likelihood of Generating MISRA C Compliant Code from MATLAB Code”

External Websites

`www.misra.org.uk`

Introduced in R2017b

coder.storageClass

Assign storage class to global variable

Syntax

```
coder.storageClass(global_name, storage_class)
```

Description

`coder.storageClass(global_name, storage_class)` assigns the storage class `storage_class` to the global variable `global_name`.

Assign the storage class to a global variable in a function that declares the global variable. You do not have to assign the storage class in more than one function.

You must have an Embedded Coder license to use `coder.storageClass`. Only when you use an Embedded Coder project or configuration object for generation of C/C++ libraries or executables does the code generation software recognize `coder.storageClass` calls.

Examples

Export Global Variables

In the function `addglobals_ex`, assign the `'ExportedGlobal'` storage class to the global variable `myglobalone` and the `'ExportedDefine'` storage class to the global variable `myglobaltwo`.

```
function y = addglobals_ex(x)

% Define the global variables.

global myglobalone;
global myglobaltwo;
```

```
% Assign the storage classes.

coder.storageClass('myglobalone','ExportedGlobal');
coder.storageClass('myglobaltwo','ExportedDefine');
y = myglobalone + myglobaltwo + x;
end
```

Create a code configuration object for a library or executable.

```
cfg = coder.config('dll','ecoder', true);
```

Generate code. This example uses the `-globals` argument to specify the types and initial values of `myglobalone` and `myglobaltwo`. Alternatively, you can define global variables in the MATLAB global workspace. To specify the type of the input argument `x`, use the `-args` option.

```
codegen -config cfg -globals {'myglobalone', 1, 'myglobaltwo', 2} -args {1} addglobals_ex -report
```

From the initial values of `1` and `2`, `codegen` determines that `myglobalone` and `myglobaltwo` have the type `double`. `codegen` defines and declares the exported variables `myglobalone` and `myglobaltwo`. It generates code that initializes `myglobalone` to `1.0` and `myglobaltwo` to `2.0`.

To view the generated code for `myglobaltwo` and `myglobalone`, click the `View report` link.

- `myglobaltwo` is defined in the `Exported data define` section in `addglobals_ex.h`.

  ```
  /* Exported data define */

  /* Definition for custom storage class: ExportedDefine */
  #define myglobaltwo                    2.0
  ```

- `myglobalone` is defined in the `Variable Definitions` section in `addglobals_ex.c`.

  ```
  /* Variable Definitions */
  /* Definition for custom storage class: ExportedGlobal */
  double myglobalone;
  ```

- `myglobalone` is declared as `extern` in the `Variable Declarations` section in `addglobals_ex.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ExportedGlobal */
extern double myglobalone;
```

- `myglobalone` is initialized in `addglobals_ex_initialize.c`.

```
#include "rt_nonfinite.h"
#include "addglobals_ex.h"
#include "addglobals_ex_initialize.h"

/* Named Constants */
#define b_myglobalone                  (1.0)

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
void addglobals_ex_initialize(void)
{
  rt_InitInfAndNaN(8U);
  myglobalone = b_myglobalone;
}
```

Import Global Variable

In the function `addglobal_im`, assign the `'ImportedExtern'` storage class to the global variable `myglobal`.

```
function y = addglobal_im(x)

% Define the global variable.

global myglobal;

% Assign the storage classes.

coder.storageClass('myglobal','ImportedExtern');
y = myglobal + x;
end
```

Create a file `c:\myfiles\myfile.c` that defines and initializes the imported variable `myglobal`.

```
#include <stdio.h>

/* Variable definitions for imported variables */
double myglobal = 1.0;
```

Create a code configuration object. Configure the code generation parameters to include `myfile.c`. For output type `'lib'`, or if you generate source code only, you can generate code without providing this file. Otherwise, you must provide this file.

```
cfg = coder.config('dll','ecoder', true);
cfg.CustomSource = 'myfile.c';
cfg.CustomInclude = 'c:\myfiles';
```

Generate the code. This example uses the `-globals` argument to specify the type and initial value of `myglobal`. Alternatively, you can define global variables in the MATLAB global workspace. For imported global variables, the code generation software uses the initial values to determine only the type.

```
codegen -config cfg -globals {'myglobal', 1} -args {1}  addglobal_im -report
```

From the initial value `1`, `codegen` determines that `myglobal` has type `double`. `codegen` declares the imported global variable `myglobal`. It does not define `myglobal` or generate code that initializes `myglobal`. `myfile.c` provides the code that defines and initializes `myglobal`.

To view the generated code for `myglobal`, click the `View report` link.

`myglobal` is declared as `extern` in the `Variable Declarations` section in `addglobal_im_data.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ImportedExtern */
extern double myglobal;
```

Import External Pointer

In the function `addglobal_imptr`, assign the `'ImportedExternPointer'` storage class to the global variable `myglobal`.

```
function y = addglobal_imptr(x)

% Define the global variable.
```

```
global myglobal;

% Assign the storage classes.

coder.storageClass('myglobal', 'ImportedExternPointer');
y = myglobal + x;
end
```

Create a file `c:\myfiles\myfile.c` that defines and initializes the imported global variable `myglobal`.

```
#include <stdio.h>

/* Variable definitions for imported variables */
double v = 1.0;
double *myglobal = &v;
```

Create a code configuration object. Configure the code generation parameters to include `myfile.c`. For output type `'lib'`, or if you generate source code only, you can generate code without providing this file. Otherwise, you must provide this file.

```
cfg = coder.config('dll','ecoder', true);
cfg.CustomSource = 'myfile.c';
cfg.CustomInclude = 'c:\myfiles';
```

Generate the code. This example uses the `-globals` argument to specify the type and initial value of the global variable `myglobal`. Alternatively, you can define global variables in the MATLAB global workspace. For imported global variables, the code generation software uses the initial values to determine only the type.

```
codegen -config cfg -globals {'myglobal', 1} -args {1} addglobal_imptr -report
```

From the initial value `1`, `codegen` determines that `myglobal` has type `double`. `codegen` declares the imported global variable `myglobal`. It does not define `myglobal` or generate code that initializes `myglobal`. `myfile.c` provides the code that defines and initializes `myglobal`.

To view the generated code for `myglobal`, click the `View report` link.

`myglobal` is declared as `extern` in the `Variable Declarations` section in `addglobal_imptr_data.h`.

```
/* Variable Declarations */
/* Declaration for custom storage class: ImportedExternPointer */
extern double *myglobal;
```

Input Arguments

`global_name` — Name of global variable

character vector

`global_name` is the name of a global variable, specified as a character vector. `global_name` must be a compile-time constant.

Example: `'myglobal'`

Data Types: `char`

`storage_class` — Name of storage class

`'ExportedGlobal'` | `'ExportedDefine'` | `'ImportedExtern'` | `'ImportedExternPointer'`

Storage class to assign to `global_var`. `storage_class` can have one of the following values.

Storage Class	Description
`'ExportedGlobal'`	• Defines the variable in the `Variable Definitions` section of the C file *`entry_point_name`*`.c`. • Declares the variable as an `extern` in the `Variable Declarations` section of the header file *`entry_point_name`*`.h` • Initializes the variable in the function *`entry_point_name`*`_initialize.h`.
`'ExportedDefine'`	Declares the variable with a `#define` directive in the `Exported data define` section of the header file *`entry_point_name`*`.h`.

Storage Class	Description
`'ImportedExtern'`	Declares the variable as an `extern` in the `Variable Declarations` section of the header file *`entry_point_name`*`_data.h`. The external code must supply the variable definition.
`'ImportedExternPointer'`	Declares the variable as an `extern` pointer in the `Variable Declarations` section of the header file *`entry_point_name`*`_data.h`. The external code must define a valid pointer variable.

- If you do not assign a storage class to a global variable, except for the declaration location, the variable behaves like it has an `'ExportedGlobal'` storage class. For an `'ExportedGlobal'` storage class, the global variable is declared in the file *`entry_point_name`*`.h`. When the global variable does not have a storage class, the variable is declared in the file *`entry_point_name`*`_data.h`.

Data Types: `char`

Limitations

- After you assign a storage class to a global variable, you cannot assign a different storage class to that global variable.
- You cannot assign a storage class to a constant global variable.

See Also

`codegen`

Topics

“Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code”
“Storage Classes for Code Generation from MATLAB Code”

Introduced in R2015b

compare

Class: cgv.CGV
Package: cgv

Compare signal data

Syntax

```
[matchNames, matchFigures, mismatchNames, mismatchFigures] =
cgv.CGV.compare(data_set1, data_set2)
[matchNames, matchFigures, mismatchNames, mismatchFigures] =
cgv.CGV.compare(data_set1, data_set2, 'Plot', param_value)
[matchNames, matchFigures, mismatchNames, mismatchFigures] =
cgv.CGV.compare(data_set1, data_set2, 'Plot', 'none', 'Signals',
signal_list, 'ToleranceFile', file_name)
```

Description

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2)` compares data from two data sets which have common signal names between both executions. Possible outputs of the `cgv.CGV.compare` function are matched signal names, figure handles to the matched signal names, mismatched signal names, and figure handles to the mismatched signal names. By default, `cgv.CGV.compare` looks at the signals which have a common name between both executions.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', param_value)` compares the signals and plots the signals according to `param_value`.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', 'none', 'Signals', signal_list, 'ToleranceFile', file_name)` compares only the given signals and does not produce plots.

Input Arguments

`data_set1, data_set2`

Output data from a model. After running the model, use the `getOutputData` function to get the data. The `cgv.CGV.getOutputData` function returns a cell array of the output signal names.

`varargin`

Variable number of parameter name and value pairs.

varargin Parameters

You can specify the following argument properties for the `cgv.CGV.compare` function using parameter name and value argument pairs. These parameters are optional.

`Plot`(optional)

Designates which comparison data to plot. The value of this parameter must be one of the following:

- `'match'`: plot the comparison of the matched signals from the two data sets
- `'mismatch'`(default): plot the comparison of the mismatched signals from the two datasets
- `'none'`: do not produce a plot

`Signals`(optional)

A cell array of character vectors, where each vector is a signal name in the output data. Use `getSavedSignals` to view the list of available signal names in the output data. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
                                    'log_data.block_name.Data(:,2)',...
                        'log_data.block_name.Data(:,3)',...
                        'log_data.block_name.Data(:,4)'};
```

If a model component contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

If `Signals` is not present, the signals are compared.

`Tolerancefile`(optional)

Name for the file created by the `createToleranceFile` function. The file contains the signal names and the associated tolerance parameter name and value pair for comparing the data.

Output Arguments

Depending on the data and the parameters, the following output arguments might be empty.

`match_names`

Cell array of matching signal names.

`match_figures`

Array of figure handles for matching signals

`mismatch_names`

Cell array of mismatching signal names

`mismatch_figures`

Array of figure handles for mismatching signals

See Also

Topics

“Verify Numerical Equivalence with CGV”

configModel

Class: cgv.Config
Package: cgv

Determine and change configuration parameter values

Syntax

`cfgObj.configModel()`

Description

`cfgObj.configModel()` determines the recommended values for the configuration parameters in the model. *`cfgObj`* is a handle to a `cgv.Config` object. The `ReportOnly` property of the object determines whether `configModel` changes the configuration parameter values.

See Also

Topics

“About Model Configurations” (Simulink)
“Programmatic Code Generation Verification”

coder.MATLABCodeTemplate class

Package: coder

Represent code generation template for MATLAB Coder

Description

Create a coder.MATLABCodeTemplate object from a code generation template (CGT) file. You can use this file to customize the code generation output for MATLAB Coder™. If a CGT file is not provided, the coder.MATLABCodeTemplate object is created from the default template file *`matlabroot`*`/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

Construction

`newObj = coder.MATLABCodeTemplate()` creates a `coder.MATLABCodeTemplate` object from the default code generation template (CGT) file *`matlabroot`*`/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

`newObj = coder.MATLABCodeTemplate(CGTFile)` creates a `coder.MATLABCodeTemplate` object from the code generation template file `CGTFile`. If the file is not on the MATLAB path, specify a full path to the file.

Input Arguments

`CGTFile`

Name of code generation template file

Methods

emitSection	Emit comments for template section
getCurrentTokens	Get current tokens
getTokenValue	Get value of token
setTokenValue	Set value of token for code generation template

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

```
newObj = coder.MATLABCodeTemplate()

newObj =

  MATLABCodeTemplate with properties:

    CGTFile: 'matlabcoder_default_template.cgt'

newObj = coder.MATLABCodeTemplate('custom_matlabcoder_template.cgt')

newObj =

  MATLABCodeTemplate with properties:

    CGTFile: 'custom_matlabcoder_template.cgt'
```

See Also

`coder.MATLABCodeTemplate.emitSection` | `coder.MATLABCodeTemplate.getCurrentTokens` | `coder.MATLABCodeTemplate.getTokenValue` | `coder.MATLABCodeTemplate.setTokenValue`

Topics

“Generate Custom File and Function Banners for C/C++ Code”
“Code Generation Template Files for MATLAB Code”

copySetup

Class: cgv.CGV
Package: cgv

Create copy of `cgv.CGV` object

Syntax

```
cgvObj2 = cgvObj1.copySetup()
```

Description

`cgvObj2 = cgvObj1.copySetup()` creates a copy of a `cgv.CGV` on page 1-61 object, *`cgvObj1`*. The copied object, `cgvObj2`, has the same configuration as *`cgvObj1`*, but does not copy results of the execution.

Examples

Make a copy of a `cgv.CGV` object, set it to run in a different mode, then run and compare the objects in a `cgv.Batch` object.

```
cgvModel = 'rtwdemo_cgv';
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');
cgvObj1.run();
cgvObj2 = cgvObj1.copySetup()
cgvObj2.setMode('sil');
cgvObj2.run();
```

Tips

- You can use this method to make a copy of a `cgv.CGV` object and then modify the object to run in a different mode by calling `setMode`.
- If you have a `cgv.CGV` object, which reported errors or failed at execution, you can use this method to copy the object and rerun it. The copied object has the same

configuration as the original object, therefore you might want to modify the location of the output files by calling `setOutputDir`. Otherwise, during execution, the copied `cgv.CGV` object overwrites the output files.

See Also

`cgv.CGV.run`

Topics

“Verify Numerical Equivalence with CGV”

copyConceptualArgsToImplementation

Copy conceptual argument specifications to implementation specifications of an entry for code replacement table entry

Syntax

```
copyConceptualArgsToImplementation(hEntry)
```

Description

`copyConceptualArgsToImplementation(hEntry)` provides a quick way to perform a shallow copy of conceptual arguments to matching implementation arguments.

The conceptual arguments and implementation arguments refer to the same argument instance. If you update an implementation argument, the corresponding conceptual argument is also updated.

Use this function when the conceptual arguments and the implementation arguments are the same for a code replacement table entry.

For arguments with an unsized type, such as `integer`, the code generator determines the size of the argument values based on hardware implementation configuration settings of the MATLAB code or model.

Examples

Copy Conceptual Argument to Implementation Arguments

This example shows how to use the `copyConceptualArgsToImplementation` function to copy conceptual argument specifications to matching implementation arguments for an addition operation.

```
hLib = RTW.TflTable;
```

```
% Create an entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg(arg);

arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg(arg);

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg(arg);

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry`* `= RTW.TflCOperationEntry`.

Example: `op_entry`

See Also

Topics

“Define Code Replacement Mappings”

“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

createAndAddConceptualArg

Create conceptual argument from specified properties and add to conceptual arguments for code replacement table entry

Syntax

```
arg = createAndAddConceptualArg(hEntry,argType, varargin)
```

Description

`arg = createAndAddConceptualArg(hEntry,argType, varargin)` creates a conceptual argument from specified properties and adds the argument to the conceptual arguments for a code replacement table entry.

Examples

Specify Conceptual Output and Input Arguments

This example shows how to use the`createAndAddConceptualArg` function to specify conceptual output and input arguments for a code replacement operator entry.

For examples of fixed-point arguments that use relative scaling or relative slope/bias values, see “Net Slope Scaling Code Replacement” and “Equal Slope and Zero Net Bias Code Replacement”.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
    'WordLength', 32, ...
```

```
    'FractionLength', 0);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT',...
    'IsSigned',     true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',         'u2', ...
    'IOType',       'RTW_IO_INPUT',...
    'IsSigned',     true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

Specify Types for Conceptual Argument

These examples show some common type specifications using `createAndAddConceptualArg`.

```
hEntry = RTW.TflCOperationEntry;
.
.
.
% uint8:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
```

```
    'DataTypeMode', 'double' );

% boolean:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );

% Fixed-point using binary-point-only scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     true, ...
    'CheckBias',      true, ...
    'DataTypeMode',   'Fixed-point: binary point scaling', ...
    'IsSigned',       true, ...
    'WordLength',     32, ...
    'FractionLength', 28);

% Fixed-point using [slope bias] scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'CheckSlope',     true, ...
    'CheckBias',      true, ...
    'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
    'IsSigned',       true, ...
    'WordLength',     16, ...
    'Slope',          15, ...
    'Bias',           2);
```

Input Arguments

hEntry — Handle to a code replacement table entry

handle

The *hEntry* is a handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`.

Example: `op_entry`

`argType` — Specifies the argument type to create
`'RTW.TflArgNumeric'` | `'RTW.TflArgMatrix'`

The *`argType`* is a character vector or string scalar that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric or `'RTW.TflArgMatrix'` for matrix.

Example: `'RTW.TflArgNumeric'`

`varargin` — Name-value pair arguments that specify the conceptual argument
name-value pair

Example: `'Name','y1'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','y1'`

`Name` — Specifies the argument name
character vector | string scalar

Example: `'Name','y1'`

`IOType` — Specifies the I/O type of the argument
`'RTW_IO_INPUT'` (default) | `'RTW_IO_OUTPUT'`

Use value `'RTW_IO_INPUT'` for input or value `'RTW_IO_OUTPUT'`.

Example: `'IOType','RTW_IO_INPUT'`

`IsSigned` — Indicates whether the argument is signed
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed.

Example: `'IsSigned',true`

`WordLength` — Specifies the word length, in bits, of the argument
`16` (default) | integer

Integer specifying the word length, in bits, of the argument. The default is `16`.

Example: `'WordLength',16`

`CheckSlope` — Selects whether to check that the slope value of the argument exactly matches the call-site slope value
`true` (default) | `false`

Boolean flag that, when set to `true` for a fixed-point argument, causes code replacement request processing to check that the slope value of the argument exactly matches the call-site slope value.

Specify `true` if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify `false` if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

Example: `'CheckSlope',true`

`CheckBias` — Selects whether to check that the bias value of the argument exactly matches the call-site bias value
`true` (default) | `false`

Boolean flag that, when set to `true` for a fixed-point argument, causes code replacement request processing to check that the bias value of the argument exactly matches the call-site bias value.

Specify `true` if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify `false` if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

Example: `'CheckBias',true`

`DataTypeMode` — Specifies the data type mode of the argument
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

`DataType` — Specifies the data type of the argument
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

`Scaling` — Specifies the data type scaling of the argument
`'BinaryPoint'` (default) | `'SlopeBias'`

Specify the data type scaling of the argument as `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

`Slope` — Specifies the slope of the argument
`1` (default) | floating-point value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters.

Example: `'Slope',1.0`

`SlopeAdjustmentFactor` — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument
`1.0` (default) | floating-point value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter.

Example: `'SlopeAdjustmentFactor',1.0`

`FixedExponent` — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument
`-15` (default) | integer value

If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter.

Example: `'FixedExponent',-15`

`Bias` — Specifies the bias of the argument
`0.0` (default) | floating-point value

Specify this parameter if you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output.

Example: `'Bias',2.0`

`FractionLength` — Specifies the fraction length for the argument
`15` (default) | integer value

Specify this parameter if you are matching a specific binary-point-only scaling combination on fixed-point operator inputs and output.

Example: `'FractionLength',15`

`BaseType` — Specifies the base data type for which a matrix argument is valid
character vector | string scalar

Example: `'BaseType','double'`

`DimRange` — Specifies the dimensions for which a matrix argument is valid
matrix dimensions

You can also specify a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means a two-dimensional matrix of size 2x2 or larger.

Example: `'DimRange',[2 2]`

Output Arguments

`arg` — Handle to the created conceptual argument
handle

The *`arg`* is a handle to the created conceptual argument. Specifying the return argument in the `createAndAddConceptualArg` function call is optional.

See Also

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

createAndAddImplementationArg

Create implementation argument from specified properties and add to implementation arguments for code replacement table entry

Syntax

```
arg = createAndAddImplementationArg(hEntry,argType,varargin)
```

Description

`arg = createAndAddImplementationArg(hEntry,argType,varargin)` creates an implementation argument from specified properties and adds the argument to the implementation arguments for a code replacement table entry.

Implementation arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, `boolean`, or `'logical'` (not fixed-point data types).

Examples

Specify Implementation Output and Input Arguments

This example shows how to use the`createAndAddImplementationArg` function with the `createAndSetCImplementationReturn` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
```

```
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

Specify Types for Implementation Argument

These examples show some common type specifications using `createAndAddImplementationArg`.

```
hEntry = RTW.TflCOperationEntry;
.
.
.
% uint8:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'u1', ...
    'IOType',         'RTW_IO_INPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
```

```
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'u1', ...
    'IOType',       'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *`hEntry`* = `RTW.TflCFunctionEntry` or *`hEntry`* = `RTW.TflCOperationEntry`.

Example: `op_entry`

`argType` — Specifies the argument type to create

`'RTW.TflArgNumeric'` | character vector | string scalar

The *`argType`* is a character vector or string scalar that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric.

Example: `'RTW.TflArgNumeric'`

`varargin` — Name-value pairs that specify the implementation argument

name-value pairs

Example: `'Name','u1'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','u1'`

`Name` — Specifies the argument name
character vector | string scalar

Example: `'Name','u1'`

`IOType` — Specifies the I/O type of the argument
`'RTW_IO_INPUT'` | character vector | string scalar

Use `'RTW_IO_INPUT'` for input.

Example: `'IOType','RTW_IO_INPUT'`

`IsSigned` — Indicates whether the argument is signed
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed.

Example: `'IsSigned',true`

`WordLength` — Specifies the word length, in bits, of the argument
16 (default) | integer value

Example: `'WordLength',16`

`DataTypeMode` — Specifies the data type mode of the argument
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

`DataType` — Specifies the data type of the argument
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

`Scaling` — Specifies the data type scaling of the argument
`'BinaryPoint'` (default) | `'SlopeBias'`

Use `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

`Slope` — Specifies the slope of the argument
`1.0` (default) | floating-point value

You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

Example: `'Slope',1.0`

`SlopeAdjustmentFactor` — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument
`1.0` (default) | floating-point value

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter, but do not specify both.

Example: `'SlopeAdjustmentFactor',1.0`

`FixedExponent` — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument
`-15` (default) | integer value

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter, but do not specify both.

Example: `'FixedExponent',0`

`Bias` — Specifies the bias of the argument
`0.0` (default) | floating-point value

Example: `'Bias',0.0`

`FractionLength` — Specifies the fraction length of the argument
`15` (default) | integer value

Example: `'FractionLength',0`

`Value` — Specifies the initial value of the argument
`0` (default) | constant value

Use this parameter only to set the value of injected constant input arguments, such as arguments that pass fraction-length values or flag values, in an implementation function signature. Do not use it for standard generated input arguments, such as `u1u2`. You can supply a constant input argument that uses this parameter anywhere in the implementation function signature, except as the return argument.

You can inject constant input arguments into the implementation signature for code replacement table entries. If the argument values or the number of arguments required depends on compile-time information, you can use custom matching. For more information, see “Customize Match and Replacement Process”.

Example: `'Value',0`

Output Arguments

`arg` — Handle to the created implementation argument

handle

Specifying the return argument in the `createAndAddImplementationArg` function call is optional.

See Also

`createAndSetCImplementationReturn`

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

createAndSetCImplementationReturn

Create implementation return argument from specified properties and add to implementation for code replacement table entry

Syntax

```
arg = createAndSetCImplementationReturn(hEntry,argType,varargin)
```

Description

`arg = createAndSetCImplementationReturn(hEntry,argType,varargin)` creates an implementation return argument from specified properties and adds the argument to the implementation for a code replacement table.

Implementation return arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, or `boolean` (not fixed-point data types).

Examples

Specify Operator Output and Input Arguments

This example shows how to use the `createAndSetCImplementationReturn` function with the `createAndAddImplementationArg` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
    'Name',       'y1', ...
    'IOType',     'RTW_IO_OUTPUT', ...
    'IsSigned',   true, ...
```

```
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u1', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
    'Name',       'u2', ...
    'IOType',     'RTW_IO_INPUT',...
    'IsSigned',   true,...
    'WordLength', 32, ...
    'FractionLength', 0 );
```

Specify Types for Operator Implementation

These examples show some common type specifications using `createAndSetCImplementationReturn`.

```
hEntry = RTW.TflCOperationEntry;
.
.
.
% uint8:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',           'y1', ...
    'IOType',         'RTW_IO_OUTPUT', ...
    'IsSigned',       false, ...
    'WordLength',     8, ...
    'FractionLength', 0 );

% single:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
```

```
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
    'Name',         'y1', ...
    'IOType',       'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'boolean' );
```

Input Arguments

`hEntry` — Handle to a code replacement table entry

handle

The *`hEntry`* is a handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *`hEntry`* = `RTW.TflCFunctionEntry` or *`hEntry`* = `RTW.TflCOperationEntry`.

Example: `op_entry`

`argType` — Specifies the argument type to create

`'RTW.TflArgNumeric'` | character vector | string scalar

The *`argType`* is a character vector or string scalar that specifies the argument type to create. Use `'RTW.TflArgNumeric'` for numeric.

Example: `'RTW.TflArgNumeric'`

`varargin` — Name-value pairs that specify the implementation return argument

name-value pairs

Example: `'Name','y1'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','y1'`

`Name` — Specifies the argument name
character vector | string scalar

Example: `'Name','y1'`

`IOType` — Specifies the I/O type of the argument
`'RTW_IO_OUTPUT'` | character vector | string scalar

Use `'RTW_IO_OUTPUT'` for output.

Example: `'IOType','RTW_IO_OUTPUT'`

`IsSigned` — Indicates whether the argument is signed
`true` (default) | `false`

Boolean value that, when set to `true`, indicates that the argument is signed. The default is `true`.

Example: `'IsSigned',true`

`WordLength` — Specifies the word length, in bits, of the argument
`16` (default) | integer

Example: `'WordLength',16`

`DataTypeMode` — Specifies the data type mode of the argument
`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'boolean'` | `'double'` | `'single'`

You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

Example: `'DataTypeMode','Fixed-point: binary point scaling'`

`DataType` — Specifies the data type of the argument
`'Fixed'` (default) | `'boolean'` | `'double'` | `'single'`

Example: `'DataType','Fixed'`

`Scaling` — Specifies the data type scaling of the argument
`'BinaryPoint'` (default) | `'SlopeBias'`

Use `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling.

Example: `'Scaling','BinaryPoint'`

`Slope` — Specifies the slope for a fixed-point argument
`1.0` (default) | floating-point value

You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

Example: `'Slope',1.0`

`SlopeAdjustmentFactor` — Specifies the slope adjustment factor (F) part of the slope, $F2^E$, of the argument
`1.0` (default) | floating-point value

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter, but do not specify both.

Example: `'SlopeAdjustmentFactor',1.0`

`FixedExponent` — Specifies the fixed exponent (E) part of the slope, $F2^E$, of the argument
`-15` (default) | integer value

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter, but do not specify both.

Example: `'FixedExponent',0`

`Bias` — Specifies the bias of the argument
`0.0` (default) | floating-point value

Example: `'Bias',0.0`

`FractionLength` — Specifies the fraction length of the argument
`15` (default) | integer value

Example: `'FractionLength',0`

Output Arguments

`arg` — Handle to the created implementation return argument
handle

Specifying the return argument in the `createAndSetCImplementationReturn` function call is optional.

See Also

`createAndAddImplementationArg`

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

createCRLEntry

Create code replacement table entry from conceptual and implementation argument string specifications

Syntax

```
tableEntry = createCRLEntry(crTable,conceptualSpecification,
implementationSpecification)
```

Description

`tableEntry = createCRLEntry(crTable,conceptualSpecification, implementationSpecification)` returns a code replacement table entry. The entry maps a conceptual representation of a function or operator to an implementation representation. The `conceptualSpecification` argument is a character vector or string scalar that defines the name and conceptual arguments, familiar to the code generator, for the function or operator to replace. The `implementationSpecification` argument is a character vector or string scalar that defines the name and C/C++ implementation arguments for the replacement function.

This function does not support:

- C++ implementations
- Data alignment
- Operator replacement with net slope arguments
- Entry parameter specifications (for example, priority, algorithm, building information)
- Semaphore and mutex function replacements

In the syntax specifications, place a space before and after an operator symbol. For example, use `double u1 + double u2` instead of `double u1+double u2`. Also, asterisk (*), tilde (~), and semicolon (;) have the following meaning.

Symbol	Meaning
*	• Following a supported data type, such as `int32*`, pass by reference (pointer). If the conceptual arguments are not scalar, in the implementation specification, pass them by reference. • As part of a fixed-point data type definition, such as `fixdt(1,32,*)`, wildcard.
~	Based on the position of the symbol, slopes or bias must be the same across data types.
;	Separates dimension ranges. For example, `[1 10; 1 100]` specifies a vector with length from 10 through 100.

The following table shows syntax for the conceptual and implementation specifications based on:

- Whether you are creating an entry for a function or operator.
- The type or characterization of the code replacement.

Type of Replacement	Conceptual Syntax	Implementation Syntax
Function Code Replacement Syntax		
Typical	`double y1 = sin(double u1)`	`double y1 = mySin(double u1)`
Derive implementation argument data types from conceptual specification	`double y1 = sin(double u1)`	`y1 = mySin(u1)`
Derive implementation arguments and data types from conceptual specification	`double y1 = sin(double u1)`	`mySin`
Change data type	`single y1 = sin(single u1)`	`double y1 = mySin(double u1)`

Type of Replacement	Conceptual Syntax	Implementation Syntax
Reorder arguments	`double y1 = atan2(double u1, double u2)`	`y1 = myAtan(u2, u1)`
Specify column vector arguments	`double y1 = sin(double u1[10])`	`double y1 = mySin(double* u1)`
Specify column vector arguments and dimension range	`double y1[1 100; 1 100] = sin(double u1[1 100; 1 100])`	`mySin(double* u1, double* y1)`
Remap return value as output argument	`double y1 = sin(double u1)`	`mySin(double u1, double* y1)`
Specify fixed-point data types	`fixdt(1,16,3) y1 = sin(fixdt(1,16,3) u1)`	`int16 y1 = mySin(int16 u1)`
Specify fixed-point data types and set `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0	`fixdt(1,16,*) y1 = sin(fixdt(1,16,*) u1)`	`int16 y1 = mySin(int16 u1)`
Specify fixed-point data types and set `SlopesMustBeTheSame` to `true`, `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0	`fixdt(1,16,~) y1 = sin(fixdt(1,16,~) u1)`	`int16 y1 = mySin(int16 u1)`

Type of Replacement	Conceptual Syntax	Implementation Syntax
Specify fixed-point data types and set `SlopesMustBeTheSame` to `true`, `BiasMustBeTheSame` to `true`, `CheckSlope` to `false`, and `CheckBias` to `false`	`fixdt(1,16,~,~) y1 = sin(fixdt(1,16,~,~) u1)`	`int16 y1 = mySin(int16 u1)`
Specify multiple output arguments	`[double y1 double y2] = foo(double u1, double u2)`	`double y1 = myFoo(double u1, double u2, double* y2)`
Operator Code Replacement Syntax		
Typical	`int16 y1 = int16 u1 + int16 u2`	`int16 y1 = myAdd(int16 u1, int16 u2)`
Specify fixed-point data types	`fixdt(1,16,3) y1 = fixdt(1,16,3) u1 + fixdt(1,16,3) u2`	`int16 y1 = myAdd(int16 u1, int16 u2)`
Specify fixed-point data types and set `CheckSlope` to `false`, `CheckBias` to `true`, and `Bias` to 0	`fixdt(1,16,*) y1 = fixdt(1,16,*) u1 + fixdt(1,16,*) u2`	`int16 y1 = myAdd(int16 u1, int16 u2)`
Specify fixed-point data types, wildcard, slopes must be the same, and zero bias	`fixdt(1,16,~,0) y1 = fixdt(1,16,~,0) u1 + fixdt(1,16,~,0) u2`	`int16 y1 = myAdd(int16 u1, int16 u2)`
Typecast	`int16 y1 = int8 u1`	`int16 y1 = myCast(int8 u1)`

Type of Replacement	Conceptual Syntax	Implementation Syntax

Shift	`int16 y1 = int16 u1 << int16 u2`
	`int16 y1 = int16 u1 >> int16 u2`
	`int16 y1 = int16 u1 .>> int16 u2`
		`int16 y1 = myShiftLeft(int16 u1, int16 u2)`
		`int16 y1 = myShiftRightArithmetic(int16 u1, int16 u2)`
		`int16 y1 = myShiftRightLogical(int16 u1, int16 u2)`

Specify relational operator	`bool y1 = int16 u1 < int16 u2`	`bool y1 = myLessThan(int6 u1, int16 u2)`
Specify multiplication and division	`int32 y1 = int32 u1 * in32 u2 / in32 u3`	`int32 y1 = myMultDiv(int32 u1, int32 u2, int32 u3)`
Specify matrix multiplication	`double y1[10][10] = double u1[10][10] * double u2[10][10]`	`myMult(double* u1, double* u2, double* y1)`
Specify element-wise matrix multiplication	`double y1[10][10] = double u1[10][10] .* double u2[10][10]`	`myMult(double* u1, double* u2, double* y1)`
Specify matrix multiplication with transpose of an input argument	`double y1[10][10] = double u1[10][10]' * double u2[10][10]`	`myMult(double* u1, double* u2, double* y1)`
Specify matrix multiplication with Hermitian of an input argument	`cdouble y1[10][10] = cdouble u1[10][10]' * cdouble u2[10][10]` `cdouble y1[10][10] = cdouble u1[10][10] * cdouble u2[10][10]'`	`myMult(cdouble* u1, cdouble* u2, cdouble* y1)`
Specify left matrix division	`double y1[10][10] = double u1[10][10] \ double u2[10][10]`	`myLeftDiv(double* u1, double* u2, double* y1)`

Type of Replacement	Conceptual Syntax	Implementation Syntax
Specify right matrix division	`double y1[10][10] = double u1[10][10] / double u2[10][10]`	`myRightDiv(double* u1, double* u2, double* y1)`

Examples

Replacement Entry for a Function

Create a table definition file that contains a function definition.

```
function crTable = crl_table_sinfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for the `sin` function.

```
tableEntry = createCRLEntry(crTable, ...
    'double y1 = sin(double u1)', ...
    'double y1 = mySin(double u1)');
```

Set entry parameters for the `sin` function. To generate the replacement code, specify that the code generator use the header and source files `mySin.h` and `mySin.c`.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Replacement Entry for an Operator

Create a table definition file that contains a function definition.

```
function crTable = crl_table_addfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for the addition operator.

```
tableEntry = createCRLEntry(crTable, ...
    'int16 y1 = int16 u1 + int16 u2', ...
    'int16 y1 = myAdd(int16 u1, int16 u2)');
```

Set entry parameters such that the entry specifies a cast-after-sum addition. To generate the replacement code, specify that the code generator use the header and source files `myAdd.h` and `myAdd.c`.

```
setTflCOperationEntryParameters(tableEntry, ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'ImplementationHeaderFile', 'myAdd.h', ...
    'ImplementationSourceFile', 'myAdd.c'));
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Replacement Entry for Fixed-Point Operator With Same Slope Across Types

Create a table definition file that contains a function definition.

```
function crTable = crl_table_intaddfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a signed fixed-point addition operation requiring the same slope across types.

```
tableEntry = createCRLEntry(crTable, ...
    'fixdt(1,16,~,0) y1 = fixdt(1,16,~,0) u1 + fixdt(1,16,~,0) u2', ...
    'int16 y1 = myAdd(int16 u1, int16 u2)');
```

Set entry parameters. Set algorithm parameters for a cast-after-sum addition and saturation and rounding modes. To generate the replacement code, specify that the code generator use the header and source files `myIntAdd.h` and `myIntAdd.c`.

```
setTflCOperationtionEntryParameters(tableEntry, ...
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode', 'RTW_ROUND_SIMPLEST', ...
    'ImplementationHeaderFile', 'myIntAdd.h', ...
    'ImplementationSourceFile', 'myIntAdd.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Replacement Entry That Assumes Implementation and Conceptual Argument Data Types Are the Same

Create a table definition file that contains a function definition.

```
function crTable = crl_table_sinfcn()
```

Within the function body, create the code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a `sin` function, where the implementation arguments are the same as the conceptual arguments.

```
tableEntry = createCRLEntry(crTable, ...
    'double y1 = sin(double u1)', ...
    'y1 = mySin(u1)');
```

Set entry parameters. To generate the replacement code, specify that the code generator use the header and source files `mySin.h` and `mySin.c`.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'ImplementationHeaderFile', 'mySin.h', ...
    'ImplementationSourceFile', 'mySin.c');
```

Add the entry to the table.

```
addEntry(crTable, tableEntry);
```

Input Arguments

`crTable` — Code replacement table

object

Table that stores one or more code replacement entries, each representing a potential replacement for a function or operator. Each entry maps a conceptual representation of a function or operator to an implementation representation and priority.

`conceptualSpecification` — Conceptual specification

character vector | string scalar

Representation of the name or symbol and conceptual input and output arguments for a function or operator that the software replaces, specified as a character vector or string scalar. Conceptual arguments observe naming conventions ('y1', 'u1', 'u2', ...) and data types familiar to the code generator. Use the syntax table in “Description” on page 1-121 to determine the syntax to use for your conceptual argument specification.

Example: `'double y1 = sin(double u1)'`

Example: `'int16 y1 = int16 u1 + int16 u2'`

`implementationSpecification` — Implementation specification

character vector | string scalar

Representation of the name and implementation input and output arguments for a C or C ++ replacement function, specified as a character vector or string scalar. Implementation arguments observe C/C++ name and data type specifications. Use the syntax table in “Description” on page 1-121 to determine the syntax for your implementation argument specification.

Example: `'double y1 = my_sin(double u1)'`

Example: `'int16 y1 = myAdd(int16 u1, int16 u2)'`

Output Arguments

`tableEntry` — Code replacement table entry

object

Code replacement table entry that represents a potential code replacement for a function or operator, returned as an object. Maps the conceptual representation of a function or operator, `conceptualSpecification`, to the C/C++ implementation representation, `implementationSpecification`.

See Also

`RTW.TflTable` | `addEntry` | `setTflCFunctionEntryParameters`

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2015a

createToleranceFile

Class: cgv.CGV
Package: cgv

Create file correlating tolerance information with signal names

Syntax

```
cgvObj.createToleranceFile(file_name , signal_list, tolerance_list)
```

Description

`cgvObj.createToleranceFile(file_name , signal_list, tolerance_list)` creates a MATLAB file, named `file_name`, containing the tolerance specification for each output signal name in `signal_list`. Each signal name in the `signal_list` corresponds to the same location of a parameter name and value pair in the `tolerance_list`.

Input Arguments

`file_name`

Name for the file containing the tolerance specification for each signal. Use this file as input to `cgv.CGV.compare` and `cgv.Batch.addTest`.

`signal_list`

A cell array of character vectors, where each vector is a signal name for data from the model. Use `getSavedSignals` to view the list of available signal names in the output data. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)',...
'log_data.block_name.Data(:,3)',...
'log_data.block_name.Data(:,4)'};
```

To specify a global tolerance for the signals, include the reserved signal name, `'global_tolerance'`, in `signal_list`. Assign a global tolerance value in the associated `tolerance_list`. If `signal_list` contains other signals, their associated tolerance value overrides the global tolerance value. In this example, the global tolerance is a relative tolerance of `0.02`.

```
signal_list = {'global_tolerance',...
'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)'};

tolerance_list = {{'relative', 0.02},...
    {'relative', 0.015},{'absolute', 0.05}};
```

Note If a model component contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if the signal name has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes in the `signal_list`. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

`tolerance_list`

Cell array of cell arrays. Each element of the outer cell array is a cell array containing a parameter name and value pair for the type of tolerance and its value. Possible parameter names are `'absolute'` | `'relative'` | `'function'`. There is a one-to-one mapping between each parameter name and value pair in the `tolerance_list` and a signal name in the `signal_list`. For example, a `tolerance_list` for a `signal_list` containing four signals might look like the following:

```
tolerance_list = {{'relative', 0.02},{'absolute', 0.06},...
    {'relative', 0.015},{'absolute', 0.05}};
```

See Also

Topics

“Verify Numerical Equivalence with CGV”

crossReleaseImport

Import generated model code from a previous release as SIL or PIL blocks

Syntax

```
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'ConfigParams',
additionalParameterList)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'DataDictionary',
dictionaryFile)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'OriginalPaths',
originalPaths,'ReplacementPaths',replacementPaths)
blockHandle = crossReleaseImport(buildFolder,
configSetOrModel,'SimulationMode',mode,'SFunctionName',
sFunctionName)
```

Description

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode)` imports previously generated model component code into the current release. The function imports the code as a cross-release block and returns the numeric handle of the block. The function displays the block in a new model window.

In an existing model, you can replace the model component with the cross-release block.

If you set `'SimulationMode'` to, for example, `'SIL'` or `'PIL'`, the function imports the code as a software-in-the-loop (SIL) or processor-in-the-loop (PIL) block. When you run a simulation or build the model, the model component uses generated code from the previous release.

To build a SIL or PIL block, the function by default uses the following parameters of the Simulink model specified by `configSetOrModel`:

- `SystemTargetFile`
- `Toolchain` or `TemplateMakefile`
- `ExistingSharedCode`
- `PortableWordSizes`
- `TargetLang`
- `TargetLangStandard`
- `TargetLibSuffix`
- `ModelReferenceNumInstancesAllowed`
- **Hardware Implementation** pane parameters

If you set `'SimulationMode'` to `'none'`, the function creates a Cross-Release Code Integration block, which:

- Supports generation of code that calls the imported code.
- Does not support normal, accelerator, or rapid accelerator mode simulations.
- Does not compile the imported code.

You can use the Cross-Release Code Integration block, for example, in workflows where compilation occurs on a different computer.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'ConfigParams', additionalParameterList)` uses additional configuration parameters for building the SIL or PIL block.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'DataDictionary', dictionaryFile)` imports generated code that uses data types specified by a data dictionary. If `configSetOrModel` is a model associated with a data dictionary, you do not have to specify the name-value pair. By default, the function identifies and uses the data dictionary when it imports the generated code. If you specify a name-value pair, the data dictionary that you specify takes precedence over the default data dictionary.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'OriginalPaths',`

`originalPaths,'ReplacementPaths',replacementPaths)` imports generated model code with relocated custom code or modified include paths. The paths specified by `replacementPaths` override the original custom code or include paths specified by `originalPaths` in a one-to-one manner. You cannot use `replacementPaths` to specify additional custom code or include paths.

`blockHandle = crossReleaseImport(buildFolder, configSetOrModel,'SimulationMode',mode,'SFunctionName', sFunctionName)` names the generated SIL or PIL block `sFunctionName_sil` or `sFunctionName_pil`. Use the `sFunctionName` argument if the default block name produces associated MATLAB identifiers that are longer than 63 characters.

Examples

Import Generated Code from Previous Release

This example shows how to import generated model code from a previous release.

Specify the location of the build folder.

```
buildFolder = fullfile(pwd,'R2015bWork', 'folderPathForP1_ert_rtw');
```

Import code for the integration model `Controller`.

```
crossReleaseImport(buildFolder,'Controller','SimulationMode','SIL');
```

The function displays a SIL block in a new Simulink editor window.

Input Arguments

`buildFolder` — Build folder

character vector

Build folder (Simulink) that contains generated model component code from a previous release.

`configSetOrModel` — Configuration object or model

`Simulink.ConfigSet`|character vector

A configuration set or Simulink model on the MATLAB path.

`mode` — Block mode
`'SIL'` | `'PIL'` | `{'SIL','PIL'}` | `'none'`

Simulation mode for block with imported code:

- `'SIL'` — Create SIL block.
- `'PIL'` — Create PIL block.
- `{'SIL','PIL'}` — Create SIL and PIL blocks.
- `'none'` — Create Cross-Release Code Integration block.

`additionalParameterList` — Additional parameters
cell array of character vectors

Additional parameters for building the SIL or PIL block.

`dictionaryFile` — Dictionary file
character vector

Data dictionary that specifies data types used by the generated code.

`originalPaths` — Original custom code folders or include paths
character vector | cell array of character vectors | string array

Folder or include paths for original custom code. Must have a one-to-one correspondence with `replacementPaths`.

`replacementPaths` — Replacement custom code folders or include paths
character vector | cell array of character vectors | string array

Folder or include paths for relocated custom code. Must have a one-to-one correspondence with `originalPaths`.

`sFunctionName` — Name for SIL or PIL block
character vector | cell array of character vectors | string array

Specify name for SIL or PIL block that contains generated code from previous release. If the default block name produces associated MATLAB identifiers that are longer than 63 characters, use this argument to specify a shorter block name.

Output Arguments

`blockHandle` — Numeric handle of a block
double|array of doubles

Numeric handle of a block. Returned as a double if `mode` is `'SIL'` or `'PIL'`. Returned as an array of doubles if `mode` is `{'SIL','PIL'}`.

See Also

`sharedCodeUpdate`

Topics

“Cross-Release Shared Utility Code Reuse”
“Cross-Release Code Integration”

Introduced in R2016b

display

Generate message that describes how to open code execution profiling report

Syntax

myExecutionProfile
myExecutionProfile.display

Description

myExecutionProfile or *myExecutionProfile*.display generates a message that describes how you can open the code execution profiling report.

myExecutionProfile is a workspace variable, specified through the configuration parameter CodeExecutionProfileVariable and generated by a simulation.

See Also

report

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

Introduced in R2011a

displayReport

Class: cgv.Config
Package: cgv

Display results of comparing configuration parameter values

Syntax

```
cfgObj.displayReport()
```

Description

`cfgObj.displayReport()` displays the results at the MATLAB Command Window of comparing the configuration parameter values for the model with the values that the object recommends. `cfgObj` is a handle to a `cgv.Config` object.

See Also

Topics

“Verify Numerical Equivalence Between Two Modes of Execution of a Model”

coder.MATLABCodeTemplate.emitSection

Class: coder.MATLABCodeTemplate
Package: coder

Emit comments for template section

Syntax

```
sectionComments = emitSection(sectionName,isCPPComment)
```

Description

`sectionComments = emitSection(sectionName,isCPPComment)` emits comments for the code template section that `sectionName` specifies. If `isCPPComment` is `true`, `emitSection` uses C++ style comments. If `emitSection` is `false`, it uses C style comments. Use `emitSection` to preview banners before you generate code. Before invoking `emitSection` to emit the banner for a template section, you must set the values for all tokens used in that section.

Input Arguments

`sectionName` — Name of templates section
character vector

Name of template section specified as one of the following values:

`'FileBanner'`	`'VariableDeclarationsBanner'`
`'FunctionBanner'`	`'VariableDefinitionsBanner'`
`'SharedUtilityBanner'`	`'FunctionDeclarationsBanner'`
`'FileTrailer'`	`'FunctionDefinitionsBanner'`
`'IncludeFilesBanner'`	`'CustomSourceCodeBanner'`
`'TypeDefinitionsBanner'`	`'CustomHeaderCodeBanner'`

`'NamedConstantsBanner'`

`isCPPComment` — C++ comment style flag
true | false

Specify `true` for C++ style comments. Specify `false` for C style comments.

Output Arguments

`sectionComments` — Comments for template section
character vector

Comments for the specified section, returned as a character vector.

Examples

Emit File Banner from Default Template

This example shows how to set the `FileName` token value and emit the default file banner.

Create a `coder.MATLABCodeTemplate` object from the default template.

```
newObj = coder.MATLABCodeTemplate
```

Set the `FileName` token value.

```
fileN = 'myfilename.c';
newObj.setTokenValue('FileName', fileN)
```

Emit the file banner.

```
newObj.emitSection('FileBanner', false)
```

The `emitSection` method generates the file banner replacing the `FileName` token with the file name that you specified. It replaces the `MATLABCoderVersion` token with the current MATLAB Coder version number. It replaces the `SourceGeneratedOn` token with the time stamp.

```
/*
 * File: myfilename.c
```

```
 *
 * MATLAB Coder version            : 2.7
 * C/C++ source code generated on  : 07-Apr-2014 17:43:32
 */
```

Emit Include Files Banner from Custom Template

This example shows how to create and modify a custom code generation template (CGT) file. It shows how to emit the include files section banner from the custom CGT file.

Create a local copy of the default CGT file for MATLAB Coder. Name it `myCGTFile.cgt`.

In your local copy of the CGT File, in the `IncludeFilesBanner` open tag, change the style to `"box"`.

```
<IncludeFilesBanner style="box">
Include Files
</IncludeFilesBanner>
```

Create a `MATLABCodeTemplate` object from your custom CGT file.

```
CGTFile = 'myCGTFile.cgt';
newObj= coder.MATLABCodeTemplate(CGTFile);
```

Emit the include files section banner using C++ style comments.

```
newObj.emitSection('IncludeFilesBanner', true)
```

The `emitSection` method generates the include files section banner using the box style with C++ style comments.

```
////////////////////////////////////////////////////////////////////////////
// Include Files                                                          //
////////////////////////////////////////////////////////////////////////////
```

See Also

`coder.MATLABCodeTemplate.getCurrentTokens` |
`coder.MATLABCodeTemplate.getTokenValue` |
`coder.MATLABCodeTemplate.setTokenValue`

Topics

“Generate Custom File and Function Banners for C/C++ Code”

“Code Generation Template Files for MATLAB Code”

enableCPP

Enable C++ support for function entry in code replacement table

Syntax

```
enableCPP(hEntry)
```

Description

`enableCPP(hEntry)` enables C++ support for a function entry in a code replacement table. This support allows you to specify a C++ namespace for the implementation function defined in the entry (see the `setNameSpace` function).

When you register a code replacement library containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the code replacement registry entry. For more information, see “Register Code Replacement Mappings”.

Examples

Enable C++ Support for Function Entry

This example shows how to use the `enableCPP` function to enable C++ support. Then, the example calls the `setNameSpace` function to set the namespace for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
    'Key',                      'sin', ...
    'Priority',                 100, ...
    'ImplementationName',       'sin', ...
    'ImplementationHeaderFile', 'cmath' );
```

```
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

Input Arguments

hEntry — Handle to a code replacement function entry

handle

The *hEntry* is a handle to a code replacement function entry previously returned by *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = *MyCustomFunctionEntry*. The *MyCustomFunctionEntry* is a class derived from `RTW.TflCFunctionEntry`.

Example: `fcn_entry`

See Also

`registerCPPFunctionEntry` | `setNameSpace`

Topics

“Math Function Code Replacement”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2010a

excludeCheck

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Exclude checks

Syntax

`excludeCheck(`*`obj, checkID`*`)`

Description

`excludeCheck(`*`obj, checkID`*`)` excludes a check from the Code Generation Advisor when a user specifies the objective. When a user selects multiple objectives, if the user specifies an additional objective that includes this check as a higher priority objective, the Code Generation Advisor displays this check.

Input Arguments

`obj`	Handle to a code generation objective object previously created.
`checkID`	Unique identifier of the check that you exclude from the new objective.

Examples

Exclude the **Identify questionable code instrumentation (data I/O)** check from the objective.

```
excludeCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

See Also

`Simulink.ModelAdvisor`

Topics

“Create Custom Code Generation Objectives”
`Simulink.ModelAdvisor`

getAlgorithmParameters

Examine algorithm parameter settings for lookup table function code replacement table entry

Syntax

```
algParams = getAlgorithmParameters(tableEntry)
```

Description

`algParams = getAlgorithmParameters(tableEntry)` returns the algorithm parameter settings for the lookup table function identified in the code replacement table entry `tableEntry`. If you call `getAlgorithmParameters` before using `setAlgorithmParameters`, `getAlgorithmParameters` lists the default parameter settings for the lookup table function.

Examples

Examine Default Parameter Settings for `prelookup` Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `prelookup` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'prelookup', ...
    'Priority', 100, ...
    'ImplementationName', 'myPrelookup');
```

Get the algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams =

  Prelookup with properties:

            ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
                 RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
       IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
       UseLastBreakpoint: [1x1 coder.algorithm.parameter.UseLastBreakpoint]
   RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
```

Examine the information for parameter `ExtrapMethod`.

```
algParams.ExtrapMethod
```

```
ans =

  ExtrapMethod with properties:

       Name: 'ExtrapMethod'
    Options: {'Linear'  'Clip'}
    Primary: 1
      Value: {'Linear'}
```

Examine the information for parameter `RndMeth`.

```
algParams.RndMeth
```

```
ans =

  RndMeth with properties:

       Name: 'RndMeth'
    Options: {1x7 cell}
    Primary: 0
      Value: {1x7 cell}
```

Examine the current `Value` setting.

```
algParams.RndMeth.Value
```

```
ans =

  Columns 1 through 6

    'Ceiling'    'Convergent'    'Floor'    'Nearest'    'Round'    'Simplest'

  Column 7

    'Zero'
```

Examine the information for parameter `IndexSearchMethod`.

```
algParams.IndexSearchMethod
```

```
ans =
```

```
  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
    Primary: 0
      Value: {'Binary search'  'Evenly spaced points'  'Linear search'}
```

Examine the information for parameter `UseLastBreakpoint`.

```
algParams.UseLastBreakpoint
```

```
ans =

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Examine the information for parameter `RemoveProtectionInput`.

```
algParams.RemoveProtectionInput
```

```
ans =

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Examine Modified Parameter Setting for `lookup2D` Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `lookup2D` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'lookup2D', ...
    'Priority', 100, ...
    'ImplementationName', 'myLookup2D');
```

Get the algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams =
```

```
  Lookup with properties:

                InterpMethod: [1x1 coder.algorithm.parameter.InterpMethod]
                ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
       UseRowMajorAlgorithm: [1x1 coder.algorithm.parameter.UseRowMajorAlgorithm]
                    RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
           IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
           UseLastTableValue: [1x1 coder.algorithm.parameter.UseLastTableValue]
      RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
  SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
    SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
            BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
```

Display the possible index search method settings.

```
algParams.IndexSearchMethod.Options
```

```
ans =

    'Linear search'    'Binary search'    'Evenly spaced points'
```

Display the current index search method setting.

```
algParams.IndexSearchMethod.Value
```

```
ans =

    'Linear search'    'Binary search'    'Evenly spaced points'
```

By default, the parameter is set to the same value set.

Set the index search method to binary search.

```
algParams.IndexSearchMethod = 'Binary search';
```

Verify the modified parameter setting.

```
algParams.IndexSearchMethod.Value
```

```
ans =

    'Binary search'
```

Input Arguments

`tableEntry` — Code replacement table entry for a lookup table function

object

Code replacement table entry that you previously created and represents a potential code replacement for a lookup table function. The entry must identify the lookup table function for which you are calling `getAlgorithmParameters`.

1. Create the entry. For example, call the function `RTW.TflCFunctionEntry`.

   ```
   tableEntry = RTW.TflCFunctionEntry;
   ```

2. Specify the name of the lookup table function for which you created the entry. Use the `Key` parameter in a call to `setTflCFunctionEntryParameters`. The following function call specifies the lookup table function `prelookup`.

   ```
   setTflCFunctionEntryParameters(tableEntry, ...
      'Key', 'prelookup', ...
      'Priority', 100, ...
      'ImplementationName', 'myPrelookup');
   ```

Output Arguments

`algParams` — Algorithm parameter settings for a lookup table function
object

Algorithm parameter settings for the lookup table function identified with the `Key` parameter in `tableEntry`.

See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable` | `addEntry` | `setAlgorithmParameters` | `setTflCFunctionEntryParameters`

Topics

“Lookup Table Function Code Replacement”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2015a

coder.dictionary.copy

Package: coder.dictionary

Copy code generation definitions between models and data dictionaries

Syntax

```
copy(sourceName,destinationName)
```

Description

`copy(sourceName,destinationName)` copies code generation definitions, such as storage classes, from the Embedded Coder Dictionary in `sourceName` to the Embedded Coder Dictionary in `destinationName`.

If a code generation definition in `sourceName` has the same name as a definition in `destinationName`, `copy` copies the source entry into the destination, and then renames the copy.

To share code definitions between models, use a Simulink data dictionary as described in "Share Embedded Coder Dictionary Definition Between Models". For general information about Embedded Coder Dictionaries and code generation definitions, see "Define Storage Classes, Memory Sections, and Function Templates for Software Architecture".

Examples

Copy Code Definitions from Model to Model

Create a storage class in the Embedded Coder Dictionary of the example model `rtwdemo_roll`. Then, copy the storage class to the Embedded Coder Dictionary in `rtwdemo_rtwecintro`.

Open the example model `rtwdemo_roll`.

```
rtwdemo_roll
```

Open the Embedded Coder Dictionary for the model. In the model, select **Code** > **C/C++ Code** > **Embedded Coder Dictionary**.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, create a storage class by clicking **Add**.

The new storage class is named `StorageClass1`.

Close the Embedded Coder Dictionary window.

Save a copy of `rtwdemo_roll` in your current folder. Saving the model saves the storage class in the Embedded Coder Dictionary.

Open the other model, `rtwdemo_rtwecintro`.

```
rtwdemo_rtwecintro
```

Copy the contents of the Embedded Coder Dictionary in `rtwdemo_roll` to the Embedded Coder Dictionary in `rtwdemo_rtwecintro`.

```
coder.dictionary.copy('rtwdemo_roll','rtwdemo_rtwecintro')
```

Open the Embedded Coder Dictionary for `rtwdemo_rtwecintro`.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the storage class `StorageClass1` appears.

Input Arguments

`sourceName` — Source model file or data dictionary

character vector

Source model file or data dictionary, specified as a character vector.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

`destinationName` — Destination model file or data dictionary
character vector

Destination model file or data dictionary, specified as a character vector.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

See Also
Embedded Coder Dictionary

Topics
“Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.dictionary.move

Package: coder.dictionary

Migrate code generation definitions between models and data dictionaries

Syntax

```
move(sourceName,destinationName)
```

Description

`move(sourceName,destinationName)` moves code generation definitions, such as storage classes, from the Embedded Coder Dictionary in `sourceName` to the Embedded Coder Dictionary in `destinationName`. The definitions are removed from `sourceName`. To copy code definitions from one Embedded Coder Dictionary to another, use `coder.dictionary.copy`.

If a code generation definition in `sourceName` has the same name as a definition in `destinationName`, `move` moves the source entry into the destination, and then renames the entry in the destination.

Use this function to:

- Move code generation definitions from a model to a Simulink data dictionary. For information about sharing code generation definitions between models by creating an Embedded Coder Dictionary in a data dictionary, see “Share Embedded Coder Dictionary Definition Between Models”.
- In a hierarchy of referenced Simulink data dictionaries, move the Embedded Coder Dictionary from one data dictionary to another. In a hierarchy of referenced dictionaries, only one dictionary can store an Embedded Coder Dictionary.

For general information about Embedded Coder Dictionaries and code generation definitions, see “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”.

Examples

Move Code Definitions from Model to Model

Create a storage class in the Embedded Coder Dictionary of the example model `rtwdemo_roll`. Then, move the storage class to the Embedded Coder Dictionary in `rtwdemo_rtwecintro`.

Open the example model `rtwdemo_roll`.

```
rtwdemo_roll
```

Open the Embedded Coder Dictionary for the model. In the model, select **Code** > **C/C++ Code** > **Embedded Coder Dictionary**.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, create a storage class by clicking **Add**.

The new storage class is named `StorageClass1`.

Close the Embedded Coder Dictionary window.

Save a copy of `rtwdemo_roll` in your current folder. Saving the model saves the storage class in the Embedded Coder Dictionary.

Open the other model, `rtwdemo_rtwecintro`.

```
rtwdemo_rtwecintro
```

Move the contents of the Embedded Coder Dictionary in `rtwdemo_roll` to the Embedded Coder Dictionary in `rtwdemo_rtwecintro`.

```
coder.dictionary.move('rtwdemo_roll','rtwdemo_rtwecintro')
```

Open the Embedded Coder Dictionary for `rtwdemo_rtwecintro`.

In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the storage class `StorageClass1` appears. The storage class no longer exists in `rtwdemo_roll`.

Input Arguments

`sourceName` — Source model file or data dictionary

character vector

Source model file or data dictionary, specified as a character vector.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

`destinationName` — Destination model file or data dictionary

character vector

Destination model file or data dictionary, specified as a character vector.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.
- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

See Also

Embedded Coder Dictionary

Topics

“Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.dictionary.remove

Package: coder.dictionary

Remove Embedded Coder Dictionary from model or Simulink data dictionary

Syntax

```
remove(sourceName)
```

Description

`remove(sourceName)` removes Embedded Coder Dictionary definitions from the model or Simulink data dictionary identified by `sourceName`. When you remove the Embedded Coder Dictionary from a model, you remove custom definitions and definitions from packages that you have loaded. The model still contains the local dictionary with definitions from the `SimulinkBuiltIn` package. When you remove the Embedded Coder Dictionary from a model that is not linked to a Simulink data dictionary with definitions, the remaining local dictionary contains definitions from the `Simulink` package. When you remove the Embedded Coder Dictionary from a Simulink data dictionary, you remove the entire Embedded Coder Dictionary, including its packages and definitions.

Use this function to:

- Remove Embedded Coder Dictionary definitions from a model.
- In a hierarchy of referenced Simulink data dictionaries, remove the Embedded Coder Dictionary from a data dictionary. In a hierarchy of referenced dictionaries, only one dictionary can store an Embedded Coder Dictionary.

To migrate code generation definitions from one source to another (for example, from a model file to a Simulink data dictionary), consider using `coder.dictionary.move`.

Examples

Remove Embedded Coder Dictionary from Model File

When you open the Embedded Coder Dictionary window for a model (see “Open the Embedded Coder Dictionary” on page 16-0) or open the Code perspective for a model, Simulink creates an Embedded Coder Dictionary in the model file. In this example, open the Embedded Coder Dictionary window for the example model `rtwdemo_roll`, create a code generation definition (a storage class), then remove the Embedded Coder Dictionary from the model.

At the command prompt, open the model.

```
rtwdemo_roll
```

In the model, select **Code** > **C/C++ Code** > **Embedded Coder Dictionary**.

The Embedded Coder Dictionary window opens, showing the contents of the new Embedded Coder Dictionary in `rtwdemo_roll`. The dictionary contains storage classes.

Click the **Add** button to create a new storage class, whose default name is `StorageClass1`.

Close the Embedded Coder Dictionary window.

At the command prompt, remove the Embedded Coder Dictionary from the model.

```
coder.dictionary.remove('rtwdemo_roll')
```

Now, the model file contains an Embedded Coder Dictionary with only the code generation definitions from the `Simulink` and `SimulinkBuiltIn` packages.

Input Arguments

`sourceName` — Target model file or data dictionary

character vector

Target model file or data dictionary, specified as a character vector.

- A model must be loaded (for example, by using `load_system`) or open.

  You do not need to specify the `.slx` file extension.

- A dictionary must be open in the Model Explorer, in the current folder, or on the MATLAB path.

  You must specify the `.sldd` file extension.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

Tip

To use `coder.dictionary.remove` on a data dictionary that references other data dictionaries, you must:

1. Temporarily remove references to dictionaries that also contain code generation definitions.
2. Use `coder.dictionary.remove` on the target dictionary.
3. Restore the dictionary references that you removed.

See Also

Embedded Coder Dictionary

Topics

“Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

getArgCategory

Class: RTW.ModelCPPArgsClass
Package: RTW

Get argument category for Simulink model port from model-specific C++ class interface

Syntax

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

Description

`category = getArgCategory(obj, portName)` gets the category — `'Value'`, `'Pointer'`, or `'Reference'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

category	Character vector specifying the argument category — `'Value'`, `'Pointer'`, or `'Reference'` — for the specified Simulink model port.

Alternatives

To view argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument categories. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getArgCategory

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

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

Description

`category = getArgCategory(obj, portName)` 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.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

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

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

getArgName

Class: RTW.ModelCPPArgsClass
Package: RTW

Get argument name for Simulink model port from model-specific C++ class interface

Syntax

`argName = getArgName(obj, portName)`

Description

`argName = getArgName(obj, portName)` gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

argName	Character vector specifying the argument name for the specified Simulink model port.

Alternatives

To view argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument names. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getArgName

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

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

Description

`argName = getArgName(obj, portName)` gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

argName	Character vector specifying the argument name for the specified Simulink model port.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

getArgPosition

Class: RTW.ModelCPPArgsClass
Package: RTW

Get argument position for Simulink model port from model-specific C++ class interface

Syntax

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

Description

`position = getArgPosition(obj, portName)` 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++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

position	Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. Without an argument for the specified port, the function returns 0.

Alternatives

To view argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument positions. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getArgPosition

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

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

Description

`position = getArgPosition(obj, portName)` 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.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

position	Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. Without an argument for the specified port, the function returns 0.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

getArgQualifier

Class: RTW.ModelCPPArgsClass
Package: RTW

Get argument type qualifier for Simulink model port from model-specific C++ class interface

Syntax

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

Description

`qualifier = getArgQualifier(obj, portName)` gets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

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

Alternatives

To view argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getArgQualifier

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

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

Description

`qualifier = getArgQualifier(obj, portName)` 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.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.

Output Arguments

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

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

getClassName

Class: RTW.ModelCPPClass
Package: RTW

Get class name from model-specific C++ class interface

Syntax

```
clsName = getClassName(obj)
```

Description

`clsName = getClassName(obj)` gets the name of the class described by the specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.

Output Arguments

clsName	A character vector specifying the name of the class described by the specified model-specific C++ class interface.

Alternatives

To view the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which displays

the model class name and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getCoderExecutionProfile

Extract execution-time profile for code generated from MATLAB function

Syntax

```
myExecutionProfile=getCoderExecutionProfile('myMATLABFunction');
```

Description

`myExecutionProfile=getCoderExecutionProfile('myMATLABFunction');` creates a workspace variable that contains the execution-time profile of the code generated from your MATLAB function.

Run the command after the completion and termination of the SIL/PIL execution of your MATLAB function.

See Also

`Sections` | `TimerTicksPerSecond` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2014b

coder.MATLABCodeTemplate.getCurrentTokens

Class: coder.MATLABCodeTemplate
Package: coder

Get current tokens

Syntax

```
currentTokens = getCurrentTokens()
```

Description

`currentTokens = getCurrentTokens()` returns list of current tokens in the MATLABCodeTemplate object

Output Arguments

`currentTokens` — Current tokens
cell array of character vectors

A list of current tokens in the MATLABCodeTemplate object, returned as a cell array of character vectors.

Examples

Create a MATLABCodeTemplate object with the default template, then list its tokens.

```
newObj = coder.MATLABCodeTemplate;
% Creates a MATLABCodeTemplate object from the default template
newObj.getCurrentTokens()
% Returns a list of tokens for the template
```

See Also

`coder.MATLABCodeTemplate.emitSection` |
`coder.MATLABCodeTemplate.getTokenValue` |
`coder.MATLABCodeTemplate.setTokenValue`

Topics

“Generate Custom File and Function Banners for C/C++ Code”
“Code Generation Template Files for MATLAB Code”

getDefaultConf

Class: RTW.ModelCPPClass
Package: RTW

Get default configuration information for model-specific C++ class interface from Simulink model

Syntax

`getDefaultConf(`*`obj`*`)`

Description

`getDefaultConf(`*`obj`*`)` initializes the specified model-specific C++ class interface to a default configuration, based on information from the ERT-based Simulink model to which the interface is attached. On the first invocation, class and step method names and step method properties are set to default values. On subsequent invocations, only step method properties are reset to default values.

Before calling this function, you must call `attachToModel`, to attach the C++ class interface to a loaded model.

Input Arguments

`obj`	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *`obj`* `= RTW.ModelCPPArgsClass on page 1-412` or *`obj`* `= RTW.ModelCPPDefaultClass on page 1-418`.

Alternatives

To view C++ class interface default configuration information in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the

Configure C++ Class Interface button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display default configuration information. In the `Default step method` view, you can see the default configuration information without clicking a button. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getDefaultConf

Class: RTW.ModelSpecificCPrototype
Package: RTW

Get default configuration information for model-specific C function prototype from Simulink model

Syntax

`getDefaultConf(`*`obj`*`)`

Description

`getDefaultConf(`*`obj`*`)` invokes the specified model-specific C function prototype to initialize the properties and the step function name of the function argument to a default configuration based on information from the ERT-based Simulink model to which it is attached. If you invoke the command again, only the properties of the function argument are reset to default values.

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

Input Arguments

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

Alternatives

Use the **Get default** button on the Configure C Step Function Interface dialog box to get the default configuration. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

getFunctionName

Class: RTW.ModelSpecificCPrototype
Package: RTW

Get function name from model-specific C function prototype

Syntax

```
fcnName = getFunctionName(obj, fcnType)
```

Description

`fcnName = getFunctionName(obj, fcnType)` gets the name of the step or initialize function described by the specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
fcnType	Optional character vector specifying which function name to get. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, gets the step function name.

Output Arguments

fcnName	A character vector specifying the name of the function described by the specified model-specific C function prototype.

See Also

Topics

“Customize Generated C Function Interfaces”

Name

Get name of profiled code section

Syntax

SectionName = *NthSectionProfile*.Name

Description

SectionName = *NthSectionProfile*.Name returns the name that identifies the profiled code section.

The software generates an identifier based on the model entity that corresponds to the profiled section of code.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

SectionName

Name that identifies profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

Name

Get name of profiled code section

Syntax

SectionName = *NthSectionProfile*.Name

Description

SectionName = *NthSectionProfile*.Name returns the name that identifies the profiled code section.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

SectionName

Name that identifies profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

Introduced in R2012b

getNamespace

Class: RTW.ModelCPPClass
Package: RTW

Get namespace from model-specific C++ class interface

Syntax

`nsName = getNamespace(obj)`

Description

`nsName = getNamespace(obj)` gets the namespace of the class described by the specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.

Output Arguments

nsName	A character vector specifying the namespace of the class described by the specified model-specific C++ class interface.

Alternatives

To view the model namespace in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which displays

the model class name and namespace and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getNumArgs

Class: RTW.ModelCPPClass
Package: RTW

Get number of step method arguments from model-specific C++ class interface

Syntax

`num = getNumArgs(obj)`

Description

`num = getNumArgs(obj)` gets the number of arguments for the step method described by the specified model-specific C++ class interface.

Input Arguments

`obj`	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.

Output Arguments

`num`	An integer specifying the number of step method arguments.

Alternatives

To view the number of step method arguments in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O`

`arguments step method` view of this dialog box, click the **Get Default Configuration** button to display the step method arguments. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getNumArgs

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

`num = getNumArgs(obj)`

Description

`num = getNumArgs(obj)` gets the number of function arguments for the function described by the specified model-specific C function prototype.

Input Arguments

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

Output Arguments

num	An integer specifying the number of function arguments.

Alternatives

Use the Configure C Step Function Interface dialog box to view C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

NumCalls

Total number of calls to profiled code section

Syntax

TotalNumCalls = *NthSectionProfile*.NumCalls

Description

TotalNumCalls = *NthSectionProfile*.NumCalls returns the total number of calls to the profiled code section over the entire simulation.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

TotalNumCalls

Total number of calls

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

“Analyze Code Execution Data”

Introduced in R2012b

NumCalls

Total number of calls to profiled code section

Syntax

TotalNumCalls = *NthSectionProfile*.NumCalls

Description

TotalNumCalls = *NthSectionProfile*.NumCalls returns the total number of calls to the profiled code section over the entire execution.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

TotalNumCalls

Total number of calls

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

Introduced in R2012b

getOutputData

Class: cgv.CGV
Package: cgv

Get output data

Syntax

```
out = cgvObj.getOutputData(InputIndex)
```

Description

`out = cgvObj.getOutputData(InputIndex)` is the method that you use to retrieve the output data that the object creates during execution of the model. `out` is the output data that the object returns. `cgvObj` is a handle to a `cgv.CGV` object. `InputIndex` is a unique numeric identifier that specifies which output data to retrieve. The `InputIndex` is associated with specific input data.

See Also

Topics

“Verify Numerical Equivalence with CGV”

getPreview

Class: RTW.ModelSpecificCPrototype
Package: RTW

Get model-specific C function prototype code preview

Syntax

```
preview = getPreview(obj, fcnType)
```

Description

`preview = getPreview(obj, fcnType)` gets the model-specific C function prototype code preview.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
fcnType	Optional. Character vector specifying which function to preview. Valid options are `'step'` and `'init'`. If *fcnType* is not specified, previews the step function.

Output Arguments

preview	Character vector specifying the function prototype for the step or initialization function.

Alternatives

Use the **C function prototype** field in the Configure C Step Function Interface dialog box to preview how your step function prototype is interpreted in generated code. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

getReportData

Class: cgv.Config
Package: cgv

Return results of comparing configuration parameter values

Syntax

```
rpt_data = cfgObj.getReportData()
```

Description

`rpt_data = cfgObj.getReportData()` compares the original configuration parameter values with the values that the object recommends. *`cfgObj`* is a handle to a `cgv.Config` object. Returns a cell array of character vectors with the model, parameter, previous value, and recommended or new value.

See Also

Topics

“Verify Numerical Equivalence with CGV”

getSavedSignals

Class: cgv.CGV
Package: cgv

Display list of signal names to command line

Syntax

```
signal_list = cgvObj.getSavedSignals(simulation_data)
```

Description

`signal_list = cgvObj.getSavedSignals(simulation_data)` returns a cell array, `signal_list`, of the output signal names of the data elements from the input data set, `simulation_data`. `simulation_data` is the output data stored in the CGV object, `cgvObj`, when you execute the model.

Tips

- After executing your model, use the `getOutputData` function to get the output data used as the input argument to the `cgvObj.getSavedSignals` function.
- Use names from the output signal list at the command line or as input arguments to other CGV functions, for example, `createToleranceFile`, `compare`, and `plot`.

See Also

Topics

"Verify Numerical Equivalence with CGV"

Number

Get number that uniquely identifies profiled code section

Syntax

SectionNumber = *NthSectionProfile*.Number

Description

SectionNumber = *NthSectionProfile*.Number returns a number that uniquely identifies the profiled code section, for example, in the code execution profiling report.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

SectionNumber

Number of profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”

“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

Number

Get number that uniquely identifies profiled code section

Syntax

SectionNumber = *NthSectionProfile*.Number

Description

SectionNumber = *NthSectionProfile*.Number returns a number that uniquely identifies the profiled code section.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

SectionNumber

Number of profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

Introduced in R2012b

pil_block_replace

Replace block in model with block from another model

Syntax

```
pil_block_replace(sourceModelBlock, destinationModelBlock)
pil_block_replace(sourceModelBlock, destinationModelBlock,
'isvisible')
```

Description

`pil_block_replace(sourceModelBlock, destinationModelBlock)` replaces a block in the destination model with a block from the source model. To preserve the original block from the destination model, in the source model, the function replaces `sourceModelBlock` with `destinationModelBlock`.

`pil_block_replace(sourceModelBlock, destinationModelBlock, 'isvisible')` highlights the replaced block in the destination model.

Examples

Replace Destination Block with Source Block

This example shows how to replace a block in a model with a block from another model.

Create a destination model that contains an Outport block, `destinationBlock`.

```
new_system('destModel')
open_system('destModel');
add_block('simulink/Sinks/Out1', 'destModel/destinationBlock')
```

Create a source model that contains a Scope block, `sourceBlock`.

```
new_system('srcModel')
open_system('srcModel');
add_block('simulink/Sinks/Scope', 'srcModel/sourceBlock')
```

Replace the Outport block in the destination model with the Scope block from the source model.

```
pil_block_replace('srcModel/sourceBlock','destModel/destinationBlock','isvisible')
```

Input Arguments

`sourceModelBlock` — Source block

character vector

Full path to the replacement block in the source model.

Example: 'srcModel/sourceBlock'

`destinationModelBlock` — Destination block

character vector

Full path to the block in the destination model, which the source block replaces.

Example: 'destModel/destinationBlock'

See Also

Topics

“Cross-Release Code Integration”

Introduced in R2006b

piltest

Verify custom target connectivity configuration for Simulink PIL simulation

Syntax

```
piltest(config)
piltest(config,'ConfigParams',additionalParameterList)
piltest(config,'TestPoint',testName)
```

Description

`piltest(config)` runs a suite of tests that verify your custom processor-in-the-loop (PIL) target connectivity configuration. In the tests, the function runs various normal, software-in-the-loop (SIL), and PIL simulations. The function compares results and produces errors if it detects differences between simulation modes. For the PIL simulations, the function extracts these parameters from `config`:

- `SystemTargetFile`
- `TargetHWDeviceType`
- `Toolchain`

In the current working folder, the function creates the `piltest` folder, which contains subfolders with test results.

`piltest(config,'ConfigParams',additionalParameterList)` extracts additional parameters from `config` for the PIL simulation.

`piltest(config,'TestPoint',testName)` runs a specific test from the test suite.

Examples

Verify Target Connectivity Configuration with `piltest`

This example uses `piltest` to verify a target connectivity configuration for PIL simulations on your development computer.

Create a target connectivity implementation in your current working folder.

```
% Make a local copy of the connectivity classes.
src_dir = ...
    fullfile(matlabroot,'toolbox','coder','simulinkcoder',...
             '+coder','+mypil');
if exist(fullfile('.','+mypil'),'dir')
    rmdir('+mypil','s')
end
mkdir +mypil
copyfile(fullfile(src_dir,'Launcher.m'), '+mypil');
copyfile(fullfile(src_dir,'TargetApplicationFramework.m'), '+mypil');
copyfile(fullfile(src_dir,'ConnectivityConfig.m'), '+mypil');

% Make the copied files writable.
fileattrib(fullfile('+mypil', '*'),'+w');

% Update the package name to reflect the new location of the files.
coder.mypil.Utils.UpdateClassName(...
    './+mypil/ConnectivityConfig.m',...
    'coder.mypil',...
    'mypil');
```

Register a target connectivity configuration using an `sl_customization.m` file. This example uses a supplied file.

```
sl_customization_path = fullfile(matlabroot,...
    'toolbox',...
    'rtw',...
    'rtwdemos',...
    'pil_demo');
addpath(sl_customization_path);
sl_refresh_customizations;
```

Specify the PIL simulation mode for the model.

```
close_system('rtwdemo_sil_topmodel')
open_system('rtwdemo_sil_topmodel')
set_param('rtwdemo_sil_topmodel','SimulationMode',...
          'processor-in-the-loop (pil)');
```

Specify the manufacturer and test hardware type. For example, PIL simulation on a 64-bit Windows® development computer requires:

```
set_param('rtwdemo_sil_topmodel','TargetHWDeviceType',...
          'Intel->x86-64 (Windows64)');
set_param('rtwdemo_sil_topmodel','TargetLongLongMode',true);
```

Run `piltest`.

```
piltest('rtwdemo_sil_topmodel', 'ConfigParam', {'ProdLongLongMode'} )
```

Input Arguments

`config` — Configuration set, configuration reference, or model

`Simulink.ConfigSet`|`Simulink.ConfigSetRef`|character vector

A configuration set, configuration set reference, or Simulink model.

`additionalParameterList` — Additional parameters

cell array of character vectors

Extract additional parameters from `config` for PIL simulation.

`testName` — Specific test

`'all'` (default) | `'verifyPILBlock'` | `'verifyModelBlock'` | `'verifyTopModel'` | `'verifyExecutionOnTarget'` | `'verifyTopModelSILPILSwitching'` | `'verifyModelBlockSILPILSwitching'`

- `'verifyPILBlock'` — For normal mode results, run a simulation of a Simulink model with a subsystem. For PIL results, replace the subsystem with a PIL block and rerun the simulation. The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyModelBlock'` — For normal mode results, run simulations of a Simulink model with a Model block in normal mode.

  For PIL mode results, run simulation loops with the Model block in PIL mode. The function varies these settings:

  - Model block parameter **Code interface** — Set to `Top model` (standalone code interface) or `Model reference`.

- **Configuration Parameters** > **Code Generation** > **Language** — Set to `C` or `C++`. For the `C++` case, the function sets **Code Generation** > **Interface** > **Code interface packaging** to `C++ class`.

The function compares normal and PIL mode results. If the function detects differences, it produces an error.

- `'verifyTopModel'` — Run simulations of a Simulink top-model in normal and PIL modes. The function compares normal and PIL mode results. If the function detects differences, it produces an error.
- `'verifyExecutionOnTarget'` — Run simulations of a Simulink model with a Model block in normal and PIL modes. For each mode, the Model block uses standalone and model reference code interfaces. For PIL mode, the function introduces a deliberate mismatch. The function compares normal and PIL mode results. If it does not detect the deliberate mismatch, it produces an error.
- `'verifyTopModelSILPILSwitching'` — For a Simulink top model:
  - Verify that production code is not regenerated when the function switches between SIL and PIL simulation modes. The function compares timestamps of the production code in each mode.
  - Compares results from SIL and PIL mode simulations to results from a normal mode simulation.

  If the function detects differences in timestamps or simulation results, it produces an error.
- `'verifyModelBlockSILPILSwitching'` — For a Simulink Model block:
  - Verify that production code is not regenerated when the Model block simulation mode switches between SIL and PIL modes. The function compares timestamps of the production code in each mode.
  - Run simulation loops with the Model block in SIL and PIL modes. The function varies the **Code interface** Model block parameter, setting this parameter to `Top model` or `Model reference`. The function compares results from SIL and PIL mode simulations to results from a normal mode simulation.

  If the function detects differences in timestamps or simulation results, it produces an error.
- `'all'` — Run all tests from the test suite.

See Also

`Simulink.ConfigSet` | `Simulink.ConfigSetRef`

Topics

“Create PIL Target Connectivity Configuration for Simulink”
“SIL and PIL Simulations”

Introduced in R2016b

piltest

Verify custom target connectivity configuration for MATLAB PIL execution

Syntax

```
piltest(config)
piltest(config,'ConfigParams',additionalParameterList)
piltest(config,'TestPoint',testName)
```

Description

`piltest(config)` runs tests that verify your custom processor-in-the-loop (PIL) target connectivity configuration. In the tests, the function runs the MATLAB function and performs PIL executions. The function compares results and produces errors if it detects differences. For PIL executions, the function extracts the `TargetHWDeviceType` and `Toolchain` settings from `config`.

In the current working folder, the function creates the `piltest` folder, which contains subfolders with test results.

`piltest(config,'ConfigParams',additionalParameterList)` extracts additional settings from `config` for the PIL execution.

`piltest(config,'TestPoint',testName)` runs the specified test.

Examples

Verify Target Connectivity Configuration with `piltest`

This example shows how you can use `piltest` to verify a target connectivity configuration for PIL execution.

Create a code generation configuration object for C/C++ static library generation.

```
cfg = coder.config('lib');
```

Create hardware configuration object, specify manufacturer and test hardware type, and assign handle to code generation object.

```
hwImpl = coder.HardwareImplementation;
hwImpl.TargetHWDeviceType = 'Atmel->AVR';
cfg.HardwareImplementation = hwImpl;
```

Specify the toolchain for code generation.

```
cfg.Toolchain = 'AVR tools for Arduino';
```

Run the function.

```
piltest(cfg)
```

Input Arguments

`config` — Configuration object
`coder.EmbeddedCodeConfig`

A configuration object that specifies code generation parameters.

`additionalParameterList` — Additional parameters
cell array of character vectors

Extract additional parameters from `config` for PIL execution.

`testName` — Specific test
`'all'` (default) | `'verifyPILConfig'`

- `'verifyPILConfig'` — For a given set of input values, the function:
  - Runs a MATLAB function on your development computer.
  - Performs PIL executions of generated MATLAB code on your target hardware with `config.TargetLang` set to `'C'` and `'C++'`.

  The function compares MATLAB function and PIL results. If the function detects differences, it produces an error.
- `'all'` — Run all tests.

See Also

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“PIL Execution of Code Generated for a Kalman Estimator”

Introduced in R2016b

Sections

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections

Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)
numberOfSections = length(myExecutionProfile.Sections)
```

Description

NthSectionProfile = *myExecutionProfile*`.Sections(`*N*`)` returns an `coder.profile.ExecutionTimeSection` object for the *N*th profiled code section.

numberOfSections = `length(`*myExecutionProfile*`.Sections)` returns the number of code sections for which profile data is available.

myExecutionProfile is a workspace variable generated by a simulation.

Input Arguments

N

Index of code section for which profile data is required

Output Arguments

NthSectionProfile

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.

- `Number` — Number of the code section.
- `NumCalls` — Number of calls to the code section.
- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire simulation.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire simulation.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire simulation.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

`numberOfSections`

Number of code sections with profile data

See Also

`TimerTicksPerSecond` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

Sections

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections

Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)
numberOfSections = length(myExecutionProfile.Sections)
```

Description

`NthSectionProfile` `= myExecutionProfile.Sections(`*`N`*`)` returns an `coder.profile.ExecutionTimeSection` object for the *`N`*th profiled code section.

`numberOfSections` `= length(myExecutionProfile.Sections)` returns the number of code sections for which profile data is available.

`myExecutionProfile` is a workspace variable that you create using `getCoderExecutionProfile`.

Input Arguments

`N`

Index of code section for which profile data is required

Output Arguments

`NthSectionProfile`

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.
- `Number` — Number of the code section.
- `NumCalls` — Number of calls to the code section.
- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire execution.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire execution.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire execution.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

`numberOfSections`

Number of code sections with profile data

See Also

`TimerTicksPerSecond` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

sharedCodeMATLABVersions

Manage MATLAB versions for cross-release code integration

Syntax

```
[registeredVersions, installationFolders] = sharedCodeMATLABVersions
sharedCodeMATLABVersions('Folder',versionInstallationFolder)
sharedCodeMATLABVersions('Remove', deregisterVersion)
```

Description

`[registeredVersions, installationFolders] = sharedCodeMATLABVersions` returns the available MATLAB versions and the installation folders.

`sharedCodeMATLABVersions('Folder',versionInstallationFolder)` registers a MATLAB version. The function specifies the folder where the MATLAB version is installed. The function checks that the folder corresponds to the `matlabroot` value for a valid installation, retrieves the MATLAB version number, and stores this information as a preference.

`sharedCodeMATLABVersions('Remove', deregisterVersion)` deregisters the MATLAB version and removes installation folder and version data.

Examples

Register Previous MATLAB Version for Cross-Release Code Integration

This code shows how you can register a previous release for your cross-release code integration workflow.

```
[registeredMATLABs, installationFolders] = sharedCodeMATLABVersions;
requiredVersion = 'R2017a';
typicalPath = 'C:\Program Files\MATLAB';
```

```
if isempty(registeredMATLABs) || ~any(strcmp(requiredVersion, registeredMATLABs))
  versionFolder = fullfile(typicalPath, requiredVersion);
  sharedCodeMATLABVersions('Folder', versionFolder);
end
```

Input Arguments

`versionInstallationFolder` — Installation folder location
character vector

Full path to the installation folder for the MATLAB version that you want to register.

Example: `'C:\Program Files\MATLAB\R2017a'`

`deregisterVersion` — Release version to deregister
character vector

MATLAB version that you want to deregister.

Example: `'R2017a'`

Output Arguments

`registeredVersions` — Registered release versions
cell array of character vectors

MATLAB release versions that are registered by the function.

`installationFolders` — Installation folder paths
cell array of character vectors

Installation folder locations for registered MATLAB versions.

See Also

`crossReleaseImport` | `sharedCodeUpdate`

Topics

“Cross-Release Shared Utility Code Reuse”
“Cross-Release Code Integration”

Introduced in R2017b

sharedCodeUpdate

Add new shared code source files to existing shared code folder

Syntax

```
sharedCodeUpdate(sourceFolder, destinationFolder)
sharedCodeUpdate(sourceFolder, destinationFolder,
'ExistingCodeSubfolder', destinationSubfolder)
sharedCodeUpdate(buildFolder, destinationFolder)
sharedCodeUpdate(buildFolder, configurationSetOrModel)
```

Description

`sharedCodeUpdate(sourceFolder, destinationFolder)` copies, for example, shared utility files from `sourceFolder` to a subfolder in `destinationFolder` provided that the files do not exist within `destinationFolder`. The function:

- Identifies files in both folders that have identical names but different content. The function does not overwrite these files in `destinationFolder`. In the Command Window, you see a `compare` link for each file. To examine differences by using the Comparison tool, click the link.
- Lists `sourceFolder` files that the function intends to copy and seeks confirmation. When you provide confirmation, the function copies the files to `destinationFolder`. By default, the destination of the copied files is a subfolder that corresponds to the release in which the files were created, for example, `R2015a` or `R2015b`.

`sharedCodeUpdate(sourceFolder, destinationFolder, 'ExistingCodeSubfolder', destinationSubfolder)` copies files to the subfolder that you specify.

`sharedCodeUpdate(buildFolder, destinationFolder)` copies shared code source files from the shared code location associated with `buildFolder`.

`sharedCodeUpdate(buildFolder, configurationSetOrModel)` copies shared code source files to the folder specified by the `'ExistingSharedCode'` parameter of a Simulink configuration set or model.

Examples

Copy Shared Utility Files to Shared Code Folder

This example shows how to copy source files from a shared utilities folder to a shared code folder.

```
sourceFolder = fullfile(pwd,'R2015bWork','slprj','ert','_sharedutils');
existingSharedCodeFolder = fullfile(pwd,'SharedUtilCode');
sharedCodeUpdate(sourceFolder, existingSharedCodeFolder);
```

Copy Shared Utility Files to Subfolder

This example shows how to copy source files from a shared utilities folder to a specified subfolder in the shared code folder.

```
sourceFolder = fullfile(pwd,'R2015bWork','slprj','ert','_sharedutils');
existingSharedCodeFolder = fullfile(pwd, 'SharedUtilCode');
destinationSubfolder = 'mySub'
sharedCodeUpdate(sourceFolder, existingSharedCodeFolder,...
'ExistingCodeSubfolder', destinationSubfolder);
```

Copy Shared Utility Files From Relocated Code Folder

This example shows how to copy shared utility files from a relocated generated code folder to an existing shared code folder.

Specify path to shared code folder that you want to update.

```
pathToExistingSharedFolder = 'C:\mySharedCodeFolder';
```

Specify the full path to the relocated generated code folder `P1_ert_rtw`.

```
anchorFolder = 'C:\myWorkFolder';
relocatedCodeFolder = fullfile(anchorFolder, 'P1_ert_rtw');
```

Update the existing shared code folder.

```
sharedCodeUpdate(relocatedCodeFolder, pathToExistingSharedFolder);
```

Input Arguments

`sourceFolder` — Source folder
character vector

File path to folder with shared code files that you want to add to existing shared code folder.

`destinationFolder` — Existing shared code folder
character vector

File path to existing shared code folder.

`destinationSubfolder` — Destination subfolder
character vector

Destination subfolder in existing shared code folder.

`buildFolder` — Build folder
character vector

Path to a build folder (Simulink) that contains previously generated model code.

`configurationSetOrModel` — Configuration set or model
character vector

Simulink configuration set or model that uses an existing shared code folder specified by the `'ExistingSharedCode'` parameter.

See Also

`crossReleaseImport`

Topics

“Cross-Release Shared Utility Code Reuse”
“Cross-Release Code Integration”

Introduced in R2016b

getStatus

Class: cgv.CGV
Package: cgv

Return execution status

Syntax

```
status = cgvObj.getStatus()
status = cgvObj.getStatus(inputName)
```

Description

`status = cgvObj.getStatus()` returns the execution status of *`cgvObj`*. *`cgvObj`* is a handle to a `cgv.CGV` object.

`status = cgvObj.getStatus(inputName)` returns the status of a single execution for `inputName`.

Input Arguments

`inputName`

`inputName` is a unique numeric or character identifier associated with input data, which is added to the `cgv.CGV` object using `addInputData`.

Output Arguments

`status`

If `inputName` is provided, `status` is the result of the execution of input data associated with `inputName`.

Value	Description
`none`	Execution has not run.
`pending`	Execution is currently running.
`completed`	Execution ran to completion without errors and output data is available.
`passed`	Baseline data was provided. Execution ran to completion and comparison to the baseline data returned no differences.
`error`	Execution produced an error.
`failed`	Baseline data was provided. Execution ran to completion and comparison to the baseline data returned a difference.

If `inputName` is not provided, the following pseudocode describes the return status:

```
if (all executions return 'passed')
    status = 'passed'
else if (all executions return 'passed' or 'completed')
  status = 'completed'
else if (an execution returns 'error')
    status = 'error'
else if (an execution returns 'failed')
    status = 'failed'
else if (an execution returns 'none' or 'pending'
    status = 'none'
```

See Also

`cgv.CGV.addBaseline` | `cgv.CGV.addInputData` | `cgv.CGV.run`

Topics

“Verify Numerical Equivalence with CGV”

getStepMethodName

Class: RTW.ModelCPPClass
Package: RTW

Get step method name from model-specific C++ class interface

Syntax

```
fcnName = getStepMethodName(obj)
```

Description

`fcnName = getStepMethodName(obj)` gets the name of the step method described by the specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.getClassInterfaceSpecification (modelName)`.

Output Arguments

fcnName	A character vector specifying the name of the step method described by the specified model-specific C++ class interface.

Alternatives

To view the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which displays

the step method name and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

getTflArgFromString

Create code replacement argument based on specified name and built-in data type

Syntax

```
arg = getTflArgFromString(hTable,name,datatype)
```

Description

`arg = getTflArgFromString(hTable,name,datatype)` creates a code replacement argument that is based on a specified name and built-in or fixed-point data type.

The `IOType` property of the created argument defaults to `'RTW_IO_INPUT'`, indicating an input argument. For an output argument, change the `IOType` value to `'RTW_IO_OUTPUT'` by directly assigning the argument property.

This function does not support matrices. To create a matrix argument, use the argument class `RTW.TflArgMatrix` as shown in “Small Matrix Operation to Processor Code Replacement”, “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”, and “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”.

Examples

Create and Add an Output Argument

This example shows how to use `getTflArgFromString` to create an `int16` output argument named `y1`. Then, the example adds the argument as a conceptual argument for a code replacement table entry.

```
hLib = RTW.TflTable;
op_entry = RTW.TflCOperationEntry;
.
.
.
```

```
arg = hLib.getTflArgFromString('y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg(arg);
```

Input Arguments

`hTable` — Handle to a code replacement table

handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* = `RTW.TflTable`.

Example: `hLib`

`name` — Specifies the name to use for a code replacement argument

character vector | string scalar

Example: `'y1'`

`datatype` — Specifies a built-in data type or a fixed-point data type to use for the code replacement argument

`'integer'` | `'int8'` | `'int16'` | `'int32'` | `'long'` | `'long_long'` | `'uinteger'` | `'uint8'` | `'uint16'` | `'uint32'` | `'ulong'` | `'ulong_long'` | `'single'` | `'double'` | `'boolean'` | `'logical'`

You can specify fixed-point data types using the `fixdt` function from Fixed-Point Designer™ software; for example, `'fixdt(1,16,2)'`.

Example: `'integer'`

Output Arguments

`arg` — Handle to the created code replacement argument

handle

The *`arg`* is a handle to the created code replacement argument, which can be specified to the `addConceptualArg` function.

See Also

`addConceptualArg`

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2008a

getTflDWorkFromString

Create code replacement DWork argument for semaphore entry based on specified name and data type

Syntax

```
arg = getTflDWorkFromString(hTable,name,datatype)
```

Description

`arg = getTflDWorkFromString(hTable,name,datatype)` creates a code replacement DWork argument, based on a specified name and data type, for a semaphore entry in a code replacement table.

Examples

Create and Add a DWork Argument

This example shows how to use the `getTflDworkFromString` to create a `void*` argument named `d1`. Then, the example adds the argument as a DWork argument for a semaphore entry in a code replacement table.

```
hLib = RTW.TflTable;

% specify semaphore init function.
hEnt = RTW.TflCSemaphoreEntry;
hEnt.setTflCSemaphoreEntryParameters( ...
  'Key',                      'RTW_SEM_INIT', ...
  'Priority',                 30, ...
  'ImplementationName',       'mySemCreate', ...
  'ImplementationHeaderFile', 'mySem.h', ...
  'ImplementationSourceFile', 'mySem.c', ...
  'ImplementationHeaderPath', LibPath, ...
  'ImplementationSourcePath', LibPath, ...
```

```
  'GenCallback',              'RTW.copyFileToBuildDir', ...
  'SideEffects',              true);

% specify conceptual operands and result
arg = hLib.getTflArgFromString('y1', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);
arg = hLib.getTflArgFromString('u1', 'void');
hEnt.addConceptualArg(arg);

% specify replacement function signature
arg=hLib.getTflArgFromString('y1','void');
hEnt.Implementation.setReturn(arg);
arg.IOType = 'RTW_IO_OUTPUT';

% DWork Arg
arg = hLib.getTflDWorkFromString('d1','void*');
hEnt.addDWorkArg(arg);
addEntry(hLib, hEnt);
```

Input Arguments

`hTable` — Handle to a code replacement table

handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* = `RTW.TflTable`.

Example: `hLib`

`name` — Specifies the name to use for the code replacement DWork argument

character vector | string scalar

Example: `'d1'`

`datatype` — Specifies a data type to use for the code replacement DWork argument

character vector | string scalar

You must specify `'void*'`.

Example: `'void*'`

Output Arguments

`arg` — Handle to the created code replacement argument
arg

The *`arg`* is a handle to the created code replacement argument, which can be specified to the `addDWorkArg` function.

See Also

`addDWorkArg`

Topics

“Semaphore and Mutex Function Replacement”
“Define Code Replacement Mappings”

Introduced in R2013a

coder.hardware

Create hardware board configuration object for C/C++ code generation from MATLAB code

Description

The `coder.hardware` function creates a `coder.Hardware` object that contains hardware board parameters for C/C++ code generation from MATLAB code.

To use a `coder.Hardware` object for code generation, assign it to the `Hardware` property of a `coder.CodeConfig` or `coder.EmbeddedCodeConfig` object that you pass to `codegen`. Assigning a `coder.Hardware` object to the `Hardware` property customizes the associated `coder.HardwareImplementation` object and other configuration parameters for the particular hardware board.

Creation

Syntax

```
coder.hardware(boardname)
coder.hardware()
```

Description

`coder.hardware(boardname)` creates a `coder.Hardware` object for the specified hardware board. The board must be supported by an installed support package. To see a list of available boards, call `coder.hardware` without input parameters.

`coder.hardware()` returns a cell array of names of boards supported by installed support packages.

Input Arguments

`boardname` — hardware board name
character vector | string scalar

Hardware board name, specified as a character vector or a string scalar.

Example: `'Raspberry Pi'`

Example: `"Raspberry Pi"`

Properties

`Name` — Name of hardware board
character vector | string scalar

Name of hardware board, specified as a character vector or a string scalar. The `coder.hardware` function sets this property using the `boardname` argument.

`CPUClockRate` — Clock rate of hardware board
100 (default) | double scalar

Clock rate of hardware board, specified as a double scalar.

Examples

Generate Code for a Supported Hardware Board

Configure code generation for a Raspberry Pi board and generate code for a function `foo`.

```
hwlist = coder.hardware();
if ismember('Raspberry Pi',hwlist)
    hw = coder.hardware('Raspberry Pi');
    cfg = coder.config('lib');
    cfg.Hardware = hw;
    codegen foo -config cfg -report
end
```

Check Supported Hardware Boards

Before creating a `coder.Hardware` object for a hardware board, check that the board is supported by an installed support package.

List all boards for which a support package is installed.

```
hwlist = coder.hardware()
```

Test for an installed support package for a particular board.

```
hwlist = coder.hardware();
if ismember('Raspberry Pi',hwlist)
    hw = coder.hardware('Raspberry Pi');
end
```

Tips

- In addition to the `Name` and `CPUClockRate` properties, a `coder.Hardware` object has dynamic properties specific to the hardware board.
- To configure code generation parameters for processor-in-the-loop (PIL) execution on a supported hardware board, use `coder.hardware`. See “PIL Execution with ARM Cortex-A at the Command Line” and “PIL Execution with ARM Cortex-A by Using the MATLAB Coder App”. PIL execution requires Embedded Coder.

See Also

`codegen` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.HardwareImplementation`

Introduced in R2015b

RTW.TflBlasEntryGenerator

Package: RTW

Create code replacement table entry for a BLAS operation

Syntax

```
obj = RTW.TflBlasEntryGenerator
```

Description

`obj = RTW.TflBlasEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a BLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for BLAS Operator

This example shows how to create a code replacement table entry for a BLAS operator, `op_entry`.

```
hTable = RTW.TflTable;

arch = computer('arch');
compilerName = 'microsoft';
LibPath = fullfile('$(MATLAB_ROOT)', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.TflBlasEntryGenerator;

libExt = 'lib';

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
```

```
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);
```

Output Arguments

`obj` — Handle to code replacement table entry for a BLAS operator

handle

The *`obj`* is a handle to the created code replacement table entry for a BLAS operator.

See Also

`RTW.TflCBlasEntryGenerator` | `RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

“Define Code Replacement Mappings”
“Matrix Multiplication Operation to MathWorks BLAS Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2010a

RTW.TflCBlasEntryGenerator

Package: RTW

Create code replacement table entry for a CBLAS operation

Syntax

```
obj = RTW.TflCBlasEntryGenerator
```

Description

`obj = RTW.TflCBlasEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a CBLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for CBLAS Operator

This example shows how to create a code replacement table entry for a CBLAS operator, `hEnt`.

```
hTable = RTW.TflTable;

arch = computer('arch');
compilerName = 'my_compiler';
LibPath = fullfile('$(MATLAB_ROOT)', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.TflCBlasEntryGenerator;

libExt = 'lib';

setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
```

```
    'Priority',                 100, ...
    'ImplementationName',       'dgemm32', ...
    'ImplementationHeaderFile', 'my_cblas_compatible_crl.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['my_lib_cblas_compatible.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',              true);
```

Output Arguments

`obj` — Handle to code replacement table entry for a CBLAS operator

handle

The *`obj`* is a handle to the created code replacement table entry for a CBLAS operator.

See Also

`RTW.TflBlasEntryGenerator` | `RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

“Define Code Replacement Mappings”
“Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2010a

RTW.TflCFunctionEntry

Package: RTW

Create code replacement table entry for a function

Syntax

```
obj = RTW.TflCFunctionEntry
```

Description

`obj = RTW.TflCFunctionEntry` creates a handle, *`obj`*, to a code replacement table entry for a function. The entry maps a conceptual representation of a function to an implementation (replacement) representation.

Examples

Create Table Entry for Function

This example shows how to create a code replacement table entry for a function, `hEnt`.

```
hEnt = RTW.TflCFunctionEntry;
```

Output Arguments

`obj` — Handle to code replacement table entry for a function

handle

The *`obj`* is a handle to the created code replacement table entry for a function.

See Also

`RTW.TflCFunctionEntryML` | `RTW.TflTable`

Topics

“Define Code Replacement Mappings”
“Math Function Code Replacement”
“Memory Function Code Replacement”
“Nonfinite Function Code Replacement”
“Lookup Table Function Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

RTW.TflCFunctionEntryML

Base class for custom code replacement table function entry

Syntax

`RTW.TflCFunctionEntryML`

Description

Derive a class from `RTW.TflCFunctionEntryML` to represent your custom function entry.

Examples

“Customize Match and Replacement Process”

See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable`

Topics

“Define Code Replacement Mappings”
“Customize Match and Replacement Process”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

RTW.TflCOperationEntry

Package: RTW

Create code replacement table entry for an operator

Syntax

```
obj = RTW.TflCOperationEntry
```

Description

`obj = RTW.TflCOperationEntry` creates a handle, *`obj`*, to a code replacement table entry for an operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for Operator

This example shows how to create a code replacement table entry for an operator, `hEnt`.

```
hEnt = RTW.TflCOperationEntry;
```

Output Arguments

`obj` — Handle to code replacement table entry for an operator

handle

The *`obj`* is a handle to the created code replacement table entry for an operator.

See Also

`RTW.TflCOperationEntryGenerator` | `RTW.TflCOperationEntryGenerator_NetSlope` | `RTW.TflCOperationEntryML` | `RTW.TflTable`

Topics

“Define Code Replacement Mappings”
“Scalar Operator Code Replacement”
“Addition and Subtraction Operator Code Replacement”
“Small Matrix Operation to Processor Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

RTW.TflCOperationEntryGenerator

Package: RTW

Create code replacement table entry for a fixed-point addition or subtraction operation

Syntax

```
obj = RTW.TflCOperationEntryGenerator
```

Description

`obj = RTW.TflCOperationEntryGenerator` creates a handle, *`obj`*, to a code replacement table entry for a fixed-point addition or subtraction operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for Fixed-Point Add or Subtract Operation

This example shows how to create a code replacement table entry for a fixed-point addition or subtraction operation, `hEnt`.

```
hEnt = RTW.TflCOperationEntryGenerator;
```

Output Arguments

`obj` — Handle to code replacement table entry for a fixed-point addition or subtraction operation

handle

The *`obj`* is a handle to the created code replacement table entry for a fixed-point addition or subtraction operation.

See Also

`RTW.TflCOperationEntry` | `RTW.TflCOperationEntryGenerator_NetSlope` | `RTW.TflCOperationEntryML` | `RTW.TflTable`

Topics

“Define Code Replacement Mappings”
“Fixed-Point Operator Code Replacement”
“Binary-Point-Only Scaling Code Replacement”
“Slope Bias Scaling Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2008a

RTW.TflCOperationEntryGenerator_NetSlope

Package: RTW

Create code replacement table entry for a net slope fixed-point operation

Syntax

```
obj = RTW.TflCOperationEntryGenerator_NetSlope
```

Description

`obj = RTW.TflCOperationEntryGenerator_NetSlope` creates a handle, *`obj`*, to a code replacement table entry for a net slope fixed-point operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Examples

Create Table Entry for Net Slope Fixed-Point Operation

This example shows how to create a code replacement table entry for a net slope fixed-point operation, `hEnt`.

```
hEnt = RTW.TflCOperationEntryGenerator_NetSlope;
```

Output Arguments

`obj` — Handle to code replacement table entry for a net slope fixed-point operation

handle

The *`obj`* is a handle to the created code replacement table entry for a net slope fixed-point operation.

See Also

`RTW.TflCOperationEntry` | `RTW.TflCOperationEntryGenerator` | `RTW.TflCOperationEntryML`

Topics

“Define Code Replacement Mappings”
“Fixed-Point Operator Code Replacement”
“Net Slope Scaling Code Replacement”
“Equal Slope and Zero Net Bias Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2008b

RTW.TflCOperationEntryML

Base class for custom code replacement table operator entry

Syntax

```
RTW.TflCOperationEntryML
```

Description

Derive a class from `RTW.TflCOperationEntryML` to represent your custom operator entry.

Examples

“Customize Code Match and Replacement for Scalar Operations”

See Also

`RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

“Define Code Replacement Mappings”
“Customize Match and Replacement Process”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

RTW.TflCSemaphoreEntry

Package: RTW

Create code replacement table entry for a semaphore or mutex

Syntax

```
obj = RTW.TflCSemaphoreEntry
```

Description

`obj = RTW.TflCSemaphoreEntry` creates a handle, *`obj`*, to a code replacement table entry for a semaphore or mutex. The entry maps a conceptual representation of a semaphore or mutex to an implementation (replacement) representation.

Examples

Create Table Entry for Semaphore or Mutex

This example shows how to create a code replacement table entry for a semaphore or mutex, `hEnt`.

```
hEnt = RTW.TflCSemaphoreEntry;
```

Output Arguments

`obj` — Handle to code replacement table entry for a semaphore or mutex

handle

The *`obj`* is a handle to the created code replacement table entry for a semaphore or mutex.

See Also

Topics

“Define Code Replacement Mappings”
“Semaphore and Mutex Function Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2010a

RTW.TflTable

Package: RTW

Create code replacement table

Syntax

```
obj = RTW.TflTable
```

Description

`obj = RTW.TflTable` creates a handle, *`obj`*, to a code replacement table.

Examples

Create a Code Replacement Table

This example shows how to create a code replacement table object, `hTable`.

```
hTable = RTW.TflTable;
```

Output Arguments

`obj` — Handle to code replacement table

handle

The *`obj`* is a handle to the created code replacement table.

See Also

Topics

“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

Time

Get simulation time for code section

Syntax

```
SimTime = NthSectionProfile.Time
```

Description

`SimTime = NthSectionProfile.Time` returns a simulation time vector that corresponds to the execution time measurements for the code section.

Examples

Get Simulation Time for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel',...
          'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel',...
          'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, get profile for the seventh code section.

```
seventhSectionProfile = executionProfile.Sections(7);
```

Get vector representing simulation time for code section.

```
simulationTimeVector = seventhSectionProfile.Time;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection`
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`SimTime` — Simulation time
double

Simulation time, in seconds, for section of code. Returned as a vector.

See Also

`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` | `Sections`

Topics

“Code Execution Profiling with SIL and PIL”
“Analyze Code Execution Data”

Introduced in R2013a

Time

Time over which code section execution time measurements are made

Syntax

```
Time = NthSectionProfile.Time
```

Description

`Time = NthSectionProfile.Time` returns a time vector corresponding to the period over which execution times are measured for the code section.

Examples

Get Time Vector for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;

codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```

```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
secondSectionProfile = executionProfile.Sections(2);
```

Get time vector for code section.

```
time = secondSectionProfile.Time;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection`
object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`Time` — Time
double

Time, in seconds, over which measurements are made for code section. Returned as a vector.

See Also

`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` | `Sections` | `getCoderExecutionProfile`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2013a

timeline

Display invocations of code sections over execution timeline

Syntax

```
timeline(executionProfile)
timeline(executionProfile, 'MaxResizeIncrement', numberOfPoints)
```

Description

`timeline(executionProfile)` displays invocations of each profiled code section over the execution timeline.

`timeline(executionProfile, 'MaxResizeIncrement', numberOfPoints)` specifies the maximum increment by which you:

- Increase the number of displayed points when you click the zoom-out tool.
- Move along the `timeline` plot when you sweep right or left with the pan tool.

Use this command when you want to review large `timeline` plots quickly.

Examples

Display Code Section Invocations

Run a simulation with a model that is configured to generate a workspace variable with execution-time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel',...
          'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel',...
          'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel',...
```

```
          'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel',...
          'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, open a code execution report.

```
report(executionProfile)
```

Under **Profiled Sections of Code**, in the **Model** column, expand all nodes. You see profile information for eight code sections. For example, the task `rtwdemo_sil_topmodel_step` and functions `CounterTypeA` and `CounterTypeB`.

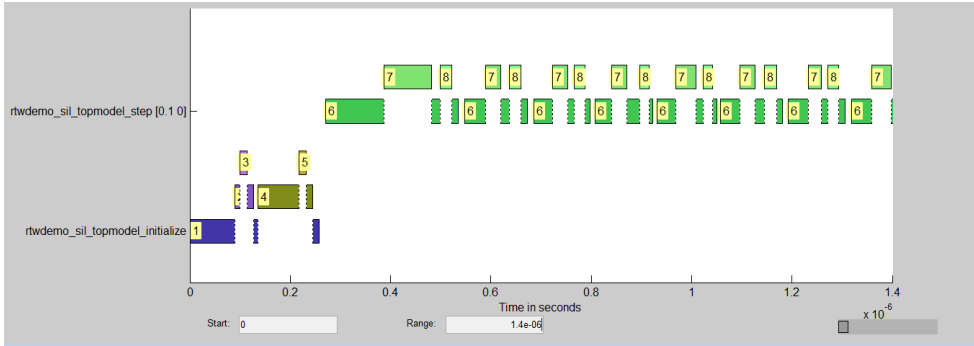
2. Profiled Sections of Code

Model	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time	Calls	
[−] rtwdemo_sil_topmodel_initialize	257	257	111	111	1	
[−] CounterTypeA	38	38	23	23	1	
CounterTypeA	15	15	15	15	1	
[−] CounterTypeB	109	109	94	94	1	
CounterTypeB	15	15	15	15	1	
[−] rtwdemo_sil_topmodel_step [0.1 0]	265	121	147	61	101	
CounterTypeA	94	36	94	36	101	
CounterTypeB	37	24	37	24	101	

Display code section invocations.

```
timeline(executionProfile)
```

In the Execution Profile window, you see numbered horizontal bars that represent invocations of the code sections.



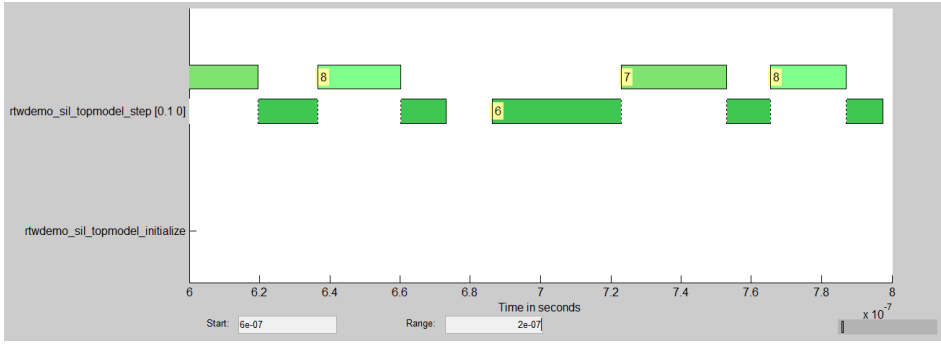
For example, the blue bars show when the first section, `rtwdemo_sil_topmodel_initialize`, is invoked.

To see the first code section, in the first row of the Code Execution Profiling Report, click the icon  .

The Code Generation Report displays the function call.

```
64          PROFILE_START_TASK_SECTION(1U);
65          rtwdemo_sil_topmodel_initialize();
66          PROFILE_END_TASK_SECTION(1U);
```

To see what code sections are invoked over a specific time period, use the **Start** and **Range** fields of the Execution Profile window. For example, in the **Start** and **Range** fields, enter `6e-07` and `2e-07` respectively. Then press **Enter**.



Between 0.6 µs and 0.8 µs, you see that the task `rtwdemo_sil_topmodel_step` (code section 6) and the functions `CounterTypeA` (code section 7) and `CounterTypeB` (code section 8) are invoked.

On the bottom right of the Execution Profile window, the indicator shows what portion of the execution timeline is being displayed.

Input Arguments

`executionProfile` — `coder.profile.ExecutionTime`
object

When you run a simulation with code execution profiling, the software generates `executionProfile` as a workspace variable.

`numberOfPoints` — Number of points
20 (default) | integer

Maximum increment for zoom-out and pan tools.

See Also

`report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

Introduced in R2013b

TimerTicksPerSecond

Get and set number of timer ticks per second

Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

Description

`timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond` returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is 10^6.

`myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal` sets the number of timer ticks per second. Use this method if the “Create PIL Target Connectivity Configuration for Simulink” does not specify this value.

`myExecutionProfile` is a workspace variable generated by a simulation.

Tip You can calculate the execution time in seconds using the formula *ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond*.

Input Arguments

timerTicksPerSecVal

Number of timer ticks per second

Output Arguments

timerTicksPerSecVal

Number of timer ticks per second

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

TimerTicksPerSecond

Get and set number of timer ticks per second

Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

Description

`timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond` returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is 10^6.

`myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal` sets the number of timer ticks per second. Use this method if the target connectivity configuration does not specify this value.

myExecutionProfile is a workspace variable that you create using `getCoderExecutionProfile`.

Tip You can calculate the execution time in seconds using the formula *ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond*.

Input Arguments

timerTicksPerSecVal

Number of timer ticks per second

Output Arguments

timerTicksPerSecVal

Number of timer ticks per second

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”
“Create PIL Target Connectivity Configuration for MATLAB”

Introduced in R2012b

coder.MATLABCodeTemplate.getTokenValue

Class: coder.MATLABCodeTemplate
Package: coder

Get value of token

Syntax

```
tokenValue = getTokenValue(tokenName)
```

Description

`tokenValue = getTokenValue(tokenName)` returns the value of the specified token.

Input Arguments

`tokenName`

Name of token

Default: empty

Output Arguments

`tokenValue` — Token value

character vector

The current value of `tokenName`, returned as a character vector.

Examples

Create a MATLABCodeTemplate object with the default template, then get the value for a token.

```
newObj = coder.MATLABCodeTemplate;
% Creates a MATLABCodeTemplate object from the default template
newObj.getCurrentTokens()
% Get list of current tokens
newObj.getTokenValue('MATLABCoderVersion')
% Check value of a token
```

See Also

`coder.MATLABCodeTemplate.emitSection` | `coder.MATLABCodeTemplate.getCurrentTokens` | `coder.MATLABCodeTemplate.setTokenValue`

Topics

“Generate Custom File and Function Banners for C/C++ Code”
“Code Generation Template Files for MATLAB Code”

ExecutionTimeInSeconds

Get execution time in seconds for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds
```

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

If you set the `CodeProfilingSaveOptions` parameter to `'SummaryOnly'`, `NthSectionProfile.ExecutionTimeInSeconds` returns an empty array. To change that parameter, open the Configuration Parameters dialog box by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to `All data`.

Examples

Get Execution Times for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;
set_param('rtwdemo_sil_topmodel', 'CodeExecutionProfiling', 'on');
set_param('rtwdemo_sil_topmodel', 'SimulationMode', 'software-in-the-loop (SIL)');
set_param('rtwdemo_sil_topmodel', 'CodeProfilingInstrumentation', 'on');
set_param('rtwdemo_sil_topmodel', 'CodeProfilingSaveOptions', 'AllData');
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, get the profile for the seventh code section.

```
SeventhSectionProfile = executionProfile.Sections(7);
```

Get vector of execution times for the code section.

```
time_vector = SeventhSectionProfile.ExecutionTimeInSeconds;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`ExecutionTimes` — Execution time measurements
double

Execution times, in seconds, for section of code. Returned as a vector.

See Also

`ExecutionTimeInTicks` | `Sections`

Topics

“Code Execution Profiling with SIL and PIL”
“Analyze Code Execution Data”

Introduced in R2013a

ExecutionTimeInSeconds

Get execution time in seconds for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds
```

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

Examples

Get Execution Times for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot,'toolbox','coder','examples','kalman');

copyfile(fullfile(src_dir,'kalman01.m'), '.')
copyfile(fullfile(src_dir,'test01_ui.m'), '.')
copyfile(fullfile(src_dir,'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir,'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;
```

```
codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');

coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
SecondSectionProfile = executionProfile.Sections(2);
```

Get vector of execution times for the code section.

```
time_vector = SecondSectionProfile.ExecutionTimeInSeconds;
```

Input Arguments

`NthSectionProfile` — `coder.profile.ExecutionTimeSection` object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`ExecutionTimes` — Execution time measurements
double

Execution times, in seconds, for section of code. Returned as a vector.

See Also

`ExecutionTimeInTicks` | `Sections` | `getCoderExecutionProfile`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2013a

ExecutionTimeInTicks

Get execution times in timer ticks for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks
```

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks` returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

If you set the `CodeProfilingSaveOptions` parameter to `'SummaryOnly'`, `NthSectionProfile.ExecutionTimeInTicks` returns an empty array. To change that parameter, open the Configuration Parameters dialog box by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to `All data`.

Tip You can calculate the execution time in seconds using the formula *ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond*

Output Arguments

ExecutionTimes

Vector of execution times, in timer ticks, for profiled section of code

SelfExecutionTimes

Vector of execution times, in timer ticks, for profiled section of code but excluding time spent in child functions

See Also

`MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

ExecutionTimeInTicks

Get execution times in timer ticks for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks
```

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks` returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Tip You can calculate the execution time in seconds using the formula *ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond*

Alternatively, set`TimerTicksPerSecond` and use `ExecutionTimeInSeconds`.

Output Arguments

ExecutionTimes

Vector of execution times, in timer ticks, for profiled section of code

See Also

`ExecutionTimeInSeconds` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` |

`MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

"Generate Execution Time Profile"
"Analyze Execution Time Data"

Introduced in R2012b

MaximumExecutionTimeCallNum

Get the call number at which maximum number of timer ticks occurred

Syntax

MaxTicksCallNum = *NthSectionProfile*.MaximumExecutionTimeCallNum

Description

MaxTicksCallNum = *NthSectionProfile*.MaximumExecutionTimeCallNum returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during a simulation.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxTicksCallNum

Call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

MaximumExecutionTimeCallNum

Get the call number at which maximum number of timer ticks occurred

Syntax

MaxTicksCallNum = *NthSectionProfile*.MaximumExecutionTimeCallNum

Description

MaxTicksCallNum = *NthSectionProfile*.MaximumExecutionTimeCallNum returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during an execution.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxTicksCallNum

Call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

MaximumExecutionTimeInTicks

Get maximum number of timer ticks for single invocation of profiled code section

Syntax

MaxTicks = *NthSectionProfile*.MaximumExecutionTimeInTicks

Description

MaxTicks = *NthSectionProfile*.MaximumExecutionTimeInTicks returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during a simulation.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxTicks

Maximum number of timer ticks for single invocation of profiled code section

See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

MaximumExecutionTimeInTicks

Get maximum number of timer ticks for single invocation of profiled code section

Syntax

MaxTicks = *NthSectionProfile*.MaximumExecutionTimeInTicks

Description

MaxTicks = *NthSectionProfile*.MaximumExecutionTimeInTicks returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during an execution.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxTicks

Maximum number of timer ticks for single invocation of profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

TotalExecutionTimeInTicks

Get total number of timer ticks recorded for profiled code section

Syntax

TotalTicks = *NthSectionProfile*.TotalExecutionTimeInTicks

Description

TotalTicks = *NthSectionProfile*.TotalExecutionTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire simulation.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

TotalTicks

Total number of timer ticks for profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”

“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

TotalExecutionTimeInTicks

Get total number of timer ticks recorded for profiled code section

Syntax

```
TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks
```

Description

`TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks` returns the total number of timer ticks recorded for the profiled code section over the entire execution.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`TotalTicks`

Total number of timer ticks for profiled code section

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

SelfTimeInTicks

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
SelfTicks = NthSectionProfile.SelfTimeInTicks
```

Description

`SelfTicks = NthSectionProfile.SelfTimeInTicks` returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`SelfTicks`

Number of timer ticks for profiled code section, excluding periods in child functions

See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

SelfTimeInTicks

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

SelfTicks = *NthSectionProfile*.SelfTimeInTicks

Description

SelfTicks = *NthSectionProfile*.SelfTimeInTicks returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

SelfTicks

Number of timer ticks for profiled code section, excluding periods in child functions

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

MaximumSelfTimeCallNum

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions

Syntax

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum

Description

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxSelfTicksCallNum

Call number at which the maximum number of self-time ticks occurred for profiled code section

See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”
“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

Introduced in R2012b

MaximumSelfTimeCallNum

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions

Syntax

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum

Description

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxSelfTicksCallNum

Call number at which the maximum number of self-time ticks occurred for profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

MaximumSelfTimeInTicks

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

MaxSelfTicks = *NthSectionProfile*.MaximumSelfTimeInTicks

Description

MaxSelfTicks = *NthSectionProfile*.MaximumSelfTimeInTicks returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxSelfTicks

Maximum number of timer ticks for profiled code section, excluding periods in child functions

See Also

ExecutionTimeInTicks | ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | display | report

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

MaximumSelfTimeInTicks

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

MaxSelfTicks = *NthSectionProfile*.MaximumSelfTimeInTicks

Description

MaxSelfTicks = *NthSectionProfile*.MaximumSelfTimeInTicks returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxSelfTicks

Maximum number of timer ticks for profiled code section, excluding periods in child functions

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumTurnaroundTimeCallNum | MaximumTurnaroundTimeInTicks | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

TotalSelfTimeInTicks

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks
```

Description

`TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks` returns the total number of timer ticks recorded for the profiled code section over the entire simulation. However, this number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TotalSelfTicks

Total number of timer ticks for profiled code section, excluding periods in child functions

See Also

`ExecutionTimeInTicks` | `ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

Introduced in R2012b

TotalSelfTimeInTicks

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks
```

Description

`TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks` returns the total number of timer ticks recorded for the profiled code section over the entire execution. However, this number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TotalSelfTicks

Total number of timer ticks for profiled code section, excluding periods in child functions

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

MaximumTurnaroundTimeInTicks

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

```
MaxTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks
```

Description

`MaxTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks` returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`MaxTurnaroundTicks`

Maximum number of timer ticks between start and finish of a single invocation of profiled code section

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

MaximumTurnaroundTimeInTicks

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

MaxTicks = *NthSectionProfile*.MaximumTurnaroundTimeInTicks

Description

MaxTicks = *NthSectionProfile*.MaximumTurnaroundTimeInTicks returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a execution. Unless the code is pre-empted, this is the same as the maximum execution time.

NthSectionProfile is a coder.profile.ExecutionTimeSection object generated by the coder.profile.ExecutionTime property Sections.

Output Arguments

MaxTurnaroundTicks

Maximum number of timer ticks between start and finish of a single invocation of profiled code section

See Also

ExecutionTimeInTicks | MaximumExecutionTimeCallNum | MaximumExecutionTimeInTicks | MaximumSelfTimeCallNum | MaximumSelfTimeInTicks | MaximumTurnaroundTimeCallNum | Name | NumCalls | Number | Sections | SelfTimeInTicks | TimerTicksPerSecond | TotalExecutionTimeInTicks | TotalSelfTimeInTicks | TotalTurnaroundTimeInTicks | TurnaroundTimeInTicks | getCoderExecutionProfile | report

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

MaximumTurnaroundTimeCallNum

Get call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

```
MaxTurnaroundTicksCallNum =
NthSectionProfile.MaximumTurnaroundTimeCallNum
```

Description

MaxTurnaroundTicksCallNum = *NthSectionProfile*.MaximumTurnaroundTimeCallNum returns the call number in which the maximum number of timer ticks was recorded between start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxTurnaroundTicksCallNum

Call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` |

`TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

MaximumTurnaroundTimeCallNum

Get call number for the code section invocation with the maximum number of timer ticks between the start and the finish

Syntax

```
MaxTurnaroundTicksCallNum =
NthSectionProfile.MaximumTurnaroundTimeCallNum
```

Description

`MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum` returns the call number in which the maximum number of timer ticks is recorded between the start and the finish of an invocation of the profiled code section. Unless the code is pre-empted, this is the same as the maximum execution time.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`MaxTurnaroundTicksCallNum`

Call number for the profiled code section invocation with the maximum number of timer ticks between start and finish

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` |

`TotalTurnaroundTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

TotalTurnaroundTimeInTicks

Get total number of timer ticks between start and finish of the profiled code section over the entire simulation.

Syntax

```
TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks
```

Description

`TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks` returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire simulation. Unless the code is pre-empted, this is the same as the total execution time.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`TotalTurnaroundTicks`

Total number of timer ticks between start and finish of the profiled code section over the entire simulation

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

TotalTurnaroundTimeInTicks

Get total number of timer ticks between start and finish of the profiled code section over the entire execution.

Syntax

```
totalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks
```

Description

`totalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks` returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire execution. Unless the code is pre-empted, this is the same as the total execution time.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`totalTurnaroundTicks`

Total number of timer ticks between start and finish of the profiled code section over the entire execution

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

TurnaroundTimeInTicks

Get number of timer ticks between start and finish of the profiled code section

Syntax

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks
```

Description

`TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks` returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`TurnaroundTicks`

Number of timer ticks between start and finish of the profiled code section

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `display` | `report`

Topics

“Code Execution Profiling with SIL and PIL”

“View and Compare Code Execution Times”
“Analyze Code Execution Data”

TurnaroundTimeInTicks

Get number of timer ticks between start and finish of the profiled code section

Syntax

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks
```

Description

`TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks` returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

`NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

`TurnaroundTicks`

Number of timer ticks between start and finish of the profiled code section

See Also

`ExecutionTimeInTicks` | `MaximumExecutionTimeCallNum` | `MaximumExecutionTimeInTicks` | `MaximumSelfTimeCallNum` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `MaximumTurnaroundTimeInTicks` | `Name` | `NumCalls` | `Number` | `Sections` | `SelfTimeInTicks` | `TimerTicksPerSecond` | `TotalExecutionTimeInTicks` | `TotalSelfTimeInTicks` | `TotalTurnaroundTimeInTicks` | `getCoderExecutionProfile` | `report`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2012b

modifyInheritedParam

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Modify inherited parameter values

Syntax

`modifyInheritedParam(`*`obj`*`, `*`paramName`*`, `*`value`*`)`

Description

`modifyInheritedParam(`*`obj`*`, `*`paramName`*`, `*`value`*`)` changes the value of an inherited parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**. Use this method when you create a new objective from an existing objective.

Input Arguments

`obj`	Handle to a code generation objective object previously created.
`paramName`	Parameter that you modify in the objective.
`value`	Value of the parameter.

Examples

Change the value of `DefaultParameterBehavior` to `Tunable` in the objective.

```
modifyInheritedParam(obj, 'DefaultParameterBehavior', 'Tunable');
```

See Also

`get_param`

Topics

“Create Custom Code Generation Objectives”

plot

Class: cgv.CGV
Package: cgv

Create plot for signal or multiple signals

Syntax

```
[signal_names, signal_figures] = cgv.CGV.plot(data_set)
[signal_names, signal_figures] = cgv.CGV.plot(data_set,'Signals',
signal_list)
```

Description

`[signal_names, signal_figures] = cgv.CGV.plot(data_set)` create a plot for each signal in the `data_set`.

`[signal_names, signal_figures] = cgv.CGV.plot(data_set,'Signals', signal_list)` create a plot for each signal in the value of `'Signals'` and return the names and figure handles for the given signal names.

Input Arguments

`data_set`

Output data from a model. After running the model, use the `getOutputData` function to get the data. The `cgv.CGV.getOutputData` function returns a cell array of the output signal names.

`'Signals', signal_list`

Parameter/value argument pair specifying the signal or signals to plot. The value for this parameter can be an individual signal name, or a cell array of character vectors, where each character vector is a signal name in the `data_set`. Use `getSavedSignals` to view

the list of available signal names in the `data_set`. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for a list of signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
               'log_data.block_name.Data(:,2)',...
               'log_data.block_name.Data(:,3)',...
               'log_data.block_name.Data(:,4)'};
```

If a component of your model contains a space or newline character, MATLAB adds parentheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

Output Arguments

Depending on the data, one or more of the following parameters might be empty:

`signal_names`

Cell array of signal names

`signal_figures`

Array of figure handles for signals

See Also

Topics

“Verify Numerical Equivalence with CGV”

register

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Register objective

Syntax

`register(obj)`

Description

`register(obj)` registers *obj* Register and add *obj* to the end of the list of available objectives that you can use with the Code Generation Advisor.

Input Arguments

obj	Handle to a code generation objective object previously created.

Examples

Register the objective:

```
register(obj);
```

See Also

Topics

“Create Custom Code Generation Objectives”
“Registering Customizations” (Simulink)

registerCFunctionEntry

Create function entry based on specified parameters and register in code replacement table

Syntax

```
entry = registerCFunctionEntry(hTable,priority,numInputs,
functionName,inputType,implementationName,outputType,headerFile,
genCallback,genFileName)
```

Description

`entry = registerCFunctionEntry(hTable,priority,numInputs, functionName,inputType,implementationName,outputType,headerFile, genCallback,genFileName)` provides a quick way to create and register a code replacement function entry.

This function can be used only if your function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*`n`*
  - For return argument, `y1`

Examples

Create C Function Entry in Table

This example shows how to use the `registerCFunctionEntry` function to create a C function entry for `sqrt` in a code replacement table.

```
hLib = RTW.TflTable;

hLib.registerCFunctionEntry(100, 1, 'sqrt', 'double', 'sqrt', ...
                            'double', '<math.h>', '', '');
```

Input Arguments

`hTable` — Handle to a code replacement table

handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* = `RTW.TflTable`.

Example: `hLib`

`priority` — Specifies the search priority of the function entry

integer 0..100

The *`priority`* specifies the search priority of the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

`numInputs` — Specifies the number of input arguments

positive integer

Example: `1`

`functionName` — Specifies the name of the function to replace

character vector | string scalar

The *`functionName`* specifies the name of the function to replace. The name must match a function name listed in “Code You Can Replace” in “What Is Code Replacement Customization?” (MATLAB code) or “What Is Code Replacement Customization?” (Simulink models).

Example: `'sqrt'`

`inputType` — Specifies the data type of the input arguments

character vector | string scalar

This function requires that the input arguments are of the same type.

Example: `'double'`

`implementationName` — Specifies the name of the implementation
character vector | string scalar

The *`implementationName`* specifies the name of the implementation. For example, if *`functionName`* is `'sqrt'`, *`implementationName`* can be `'sqrt'` or a different name.

Example: `'sqrt'`

`outputType` — Specifies the data type of the return argument
character vector | string scalar

Example: `'double'`

`headerFile` — Specifies the header file that declares the implementation function
character vector | string scalar

Example: `'<math.h>'`

`genCallback` — Specifies callback that follows code generation
`''` | `'RTW.copyFileToBuildDir'`

The *`genCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `''`

`genFileName` — Specifies `''`
`''` | character vector | string scalar

This argument is reserved for MathWorks developers.

Example: `''`

Output Arguments

`entry` — Handle to the created code replacement function entry
handle

The *`entry`* is a handle to the created code replacement function entry. Specifying the return argument in the `registerCFunctionEntry` function call is optional.

See Also

`registerCPromotableMacroEntry`

Topics

“Define Code Replacement Mappings”

Introduced in R2007b

registerCPPFunctionEntry

Create C++ function entry based on specified parameters and register in code replacement table

Syntax

Description

provides a quick way to create and register a code replacement C++ function entry.

This function can be used only if your C++ function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*`n`*
  - For return argument, `y1`

When you register a code replacement library containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the library registry entry. For more information, see “Register Code Replacement Mappings”.

Examples

Create C++ Function Entry in Table

This example shows how to use the `registerCPPFunctionEntry` function to create a C ++ function entry for `sin` in a code replacement table.

```
hLib = RTW.TflTable;
```

```
hLib.registerCPPFunctionEntry(100, 1, 'sin', 'single', 'sin', ...
                              'single', 'cmath', '', '', 'std');
```

Input Arguments

`hTable` — Handle to a code replacement table

handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* = `RTW.TflTable`.

Example: `hLib`

`priority` — Specifies the search priority for the function entry

integer 0..100

The *`priority`* specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

`numInputs` — Specifies the number of input arguments

positive integer

Example: `1`

`functionName` — Specifies the name of the function to replace

character vector | string scalar

The *`functionName`* specifies the name of the function to replace. The name must match a function listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'sin'`

`inputType` — Specifies the data type of the input arguments

character vector | string scalar

This function requires that the input arguments are of the same type.

Example: `'double'`

`implementationName` — Specifies the name of the implementation
character vector | string scalar

The *`implementationName`* specifies the name of the implementation. For example, if *`functionName`* is `'sqrt'`, *`implementationName`* can be `'sqrt'` or a different name.

Example: `'sqrt'`

`outputType` — Specifies the data type of the return argument
character vector | string scalar

Example: `'double'`

`headerFile` — Specifies the header file that declares the implementation function
character vector | string scalar

Example: `'<math.h>'`

`genCallback` — Specifies callback that follows code generation
`''` | `'RTW.copyFileToBuildDir'`

The *`genCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `''`

`genFileName` — Specifies `''`
`''`

This argument is reserved for MathWorks developers.

Example: `''`

`nameSpace` — Specifies the C++ namespace in which the implementation function is defined
character vector | string scalar

The *`nameSpace`* specifies the C++ namespace in which the implementation function is defined. If this function entry is matched, the software emits the namespace in the

generated function code (for example, `std::sin(tfl_cpp_U.In1)`). If you specify `''`, the software does not emit a namespace designation in the generated code.

Example: `'std'`

Output Arguments

`entry` — Handle to the created C++ function entry
handle

The *`entry`* is a handle to the created C++ function entry. Specifying the return argument in the `registerCPPFunctionEntry` function call is optional.

See Also

`enableCPP` | `setNameSpace`

Topics

"Define Code Replacement Mappings"

Introduced in R2010a

registerCPromotableMacroEntry

Create promotable code replacement macro entry based on specified parameters and register in code replacement table (for `abs` function replacement only)

Syntax

```
entry = registerCPromotableMacroEntry(hTable,priority,numInputs,
functionName,inputType,implementationName,outputType,headerFile,
genCallback,genFileName)
```

Description

`entry = registerCPromotableMacroEntry(hTable,priority,numInputs, functionName,inputType,implementationName,outputType,headerFile, genCallback,genFileName)` creates a promotable macro entry based on specified parameters and registers the entry in the code replacement table. A promotable macro entry promotes the output data type based on the target word size.

This function provides a quick way to create and register a promotable macro entry. This function can be used only if your code replacement function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*`n`*
  - For return argument, `y1`

Use this function only for `abs` function replacement. For other functions supported for replacement, use `registerCFunctionEntry`.

Examples

Create Promotable Macro Entry in Table

This example shows how to use the `registerCPromotableMacroEntry` function to create a promotable macro entry for `abs` in a code replacement table.

```
hLib = RTW.TflTable;

hLib.registerCPromotableMacroEntry(100, 1, 'abs', ...
    'double', 'abs_prime', ...
    'double', '<math_prime.h>', '', '');
```

Input Arguments

`hTable` — Handle to a code replacement table

handle

The *`hTable`* is a handle to a code replacement table previously returned by *`hTable`* = `RTW.TflTable`.

Example: `hLib`

`priority` — Specifies the search priority for the function entry

integer 0..100

The *`priority`* specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `100`

`numInputs` — Specifies the number of input arguments

positive integer

Example: `1`

`functionName` — Specifies the name of the function to replace

character vector | string scalar

The *`functionName`* specifies the name of the function to be replaced. Specify `'abs'`. Use this function only for `abs` function replacement.

Example: `'abs'`

`inputType` — Specifies the data type of the input arguments
character vector | string scalar

This function requires that the input arguments are of the same type.

Example: `'double'`

`implementationName` — Specifies the name of the implementation
character vector | string scalar

The *`implementationName`* specifies the name of the implementation. For example, assuming *`functionName`* is `'abs'`, *`implementationName`* can be `'abs'` or a different name of your choosing.

Example: `'abs'`

`outputType` — Specifies the data type of the return argument
character vector | string scalar

Example: `'double'`

`headerFile` — Specifies the header file that declares the implementation function
character vector | string scalar

Example: `'<math.h>'`

`genCallback` — Specifies callback that follows code generation
`''` | `'RTW.copyFileToBuildDir'`

The *`genCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `''`

`genFileName` — Specifies `''`
`''` | character vector | string scalar

This argument is reserved for MathWorks developers.

Example: `''`

Output Arguments

`entry` — Handle to the created promotable macro entry
handle

The *`entry`* is a handle to the created promotable macro entry. Specifying the return argument in the `registerCPromotableMacroEntry` function call is optional.

See Also

`registerCFunctionEntry`

Topics

“Define Code Replacement Mappings”

Introduced in R2007b

removeInheritedCheck

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Remove inherited checks

Syntax

`removeInheritedCheck(`*`obj`*`, `*`checkID`*`)`

Description

`removeInheritedCheck(`*`obj`*`, `*`checkID`*`)` removes an inherited check from the objective definition. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

Input Arguments

`obj`	Handle to a code generation objective object previously created.
`checkID`	Unique identifier of the check that you remove from the new objective.

Examples

Remove the **Identify questionable code instrumentation (data I/O)** check from the objective.

```
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

)

See Also

`Simulink.ModelAdvisor`

Topics

"Create Custom Code Generation Objectives"
`Simulink.ModelAdvisor`

removeInheritedParam

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Remove inherited parameters

Syntax

`removeInheritedParam(`*`obj`*`, `*`paramName`*`)`

Description

`removeInheritedParam(`*`obj`*`, `*`paramName`*`)` removes an inherited parameter from this objective. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another objective includes the parameter, the Code Generation Advisor reviews the parameter value using **Check model configuration settings against code generation objectives**.

Input Arguments

`obj`	Handle to a code generation objective object previously created.
`paramName`	Parameter that you want to remove from the objective.

Examples

Remove `DefaultParameterBehavior` from the objective.

```
removeInheritedParam(obj, 'DefaultParameterBehavior');
```

See Also

`get_param`

Topics

“Create Custom Code Generation Objectives”

report

Open code execution profiling report and specify display of time measurements.

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)
report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor',
'1e-06', 'NumericFormat', '%0.3f')
```

Description

When you run a SIL or PIL simulation with code execution profiling, the software generates the workspace variable *`myExecutionProfile`*, specified in **Configuration Parameters** > **Code Generation** > **Verification** > **Workspace variable**.

`report(`*`myExecutionProfile`*`)` opens the code execution profiling report using default display options.

`report(`*`myExecutionProfile`*`, Name1, Value1, Name2, Value2, ...)` opens the report with display options specified by the name-value *character vector* pairs.

`report(`*`myExecutionProfile`*`, 'Units', 'Seconds', 'ScaleFactor', '1e-06', 'NumericFormat', '%0.3f')` displays time in microseconds (10^{-6} seconds) with a precision of three decimal places.

Name-Value Pair	Details
`'Units'`, `'Seconds'` or `'Units'`, `'Ticks'`	Time measurements displayed in seconds or timer ticks. Default: • SIL simulation on Windows — Seconds • SIL simulation on non-Windows — Timer ticks • PIL simulation — Seconds, if number of timer ticks per second has been specified by the target connectivity configuration. Otherwise, ticks.
`'ScaleFactor'`, *`Value`*	Scale factor for displayed measurements. For example, to display measurements in microseconds, use the name-value pair `'ScaleFactor'`, `'1e-6'`. *`Value`* must be a character vector representation of a number that is a power of 10. For example, `'1'`, `'1e-6'`, or `'1e-9'`. Default value is `'1e-9'`. To specify the scale factor, you must also specify `'Units'`, `'Seconds'`.
`'NumericFormat'`, *`Convention`*	Numeric format for displayed measurements. Use the *decimal* convention utilized by the ANSI® C function `sprintf`, for example, `'%1.2f'`. Default is `'%0.0f'`. To specify the numeric format, you must also specify `'Units'`, `'Seconds'`.

See Also

`annotate` | `display`

Topics

“Code Execution Profiling with SIL and PIL”
“View and Compare Code Execution Times”

Introduced in R2011b

report

Open code execution profiling report and specify display of time measurements.

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)
report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor',
'1e-06', 'NumericFormat', '%0.3f')
```

Description

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)` opens the report with display options specified by the name-value *character vector* pairs.

`report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor', '1e-06', 'NumericFormat', '%0.3f')` displays time in microseconds (10^{-6} seconds) with a precision of three decimal places.

`myExecutionProfile` is a workspace variable that you create using `getCoderExecutionProfile`.

Name-Value Pair	Details
`'Units'`, `'Seconds'` or `'Units'`, `'Ticks'`	Time measurements displayed in seconds or timer ticks. Default: • SIL execution on Windows — Seconds • SIL execution on non-Windows — Timer ticks • PIL execution — Seconds, if number of timer ticks per second has been specified by the target connectivity configuration. Otherwise, ticks.
`'ScaleFactor'`, *`Value`*	Scale factor for displayed measurements. For example, to display measurements in microseconds, use the name-value pair `'ScaleFactor'`, `'1e-6'`. *`Value`* must be a character vector representation of a number that is a power of 10. For example, `'1'`, `'1e-6'`, or `'1e-9'`. Default value is `'1e-9'`. To specify the scale factor, you must also specify `'Units'`, `'Seconds'`.
`'NumericFormat'`, *`Convention`*	Numeric format for displayed measurements. Use the *decimal* convention utilized by the ANSI C function `sprintf`, for example, `'%1.2f'`. Default is `'%0.0f'`. To specify the numeric format, you must also specify `'Units'`, `'Seconds'`.

See Also

`Sections` | `TimerTicksPerSecond` | `getCoderExecutionProfile`

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

Introduced in R2011b

rtIOStreamClose

Shut down communications channel

Syntax

```
errFlg = rtIOStreamClose(streamID)
```

Description

`errFlg = rtIOStreamClose(streamID)` shuts down the communications channel and cleans up associated resources.

Examples

Close Communications Channel

This code from `rtiostreamtest.c` detects errors when closing the communications channel.

```
static int closeServer(void)
{
    const int errorOccurred = rtIOStreamClose(streamID);
    if (errorOccurred == RTIOSTREAM_ERROR)
    {
        return errorOccurred;
    }
    return RTIOSTREAM_NO_ERROR;
}
```

Input Arguments

`streamID` — Stream handle
scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

Output Arguments

`errFlg` — Error flag
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtiostream_wrapper`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation”
“Configure Processor-In-The-Loop (PIL) for a Custom Target”

Introduced in R2009a

rtIOStreamOpen

Initialize communications channel

Syntax

```
streamID = rtIOStreamOpen(argCount, argValues)
```

Description

`streamID = rtIOStreamOpen(argCount, argValues)` initializes a communication stream to allow the exchange of data between the development computer and target processor.

Examples

Initialize Communications Channel

This code from `rtiostreamtest.c` initializes a communication stream and checks for errors.

```
static int openServer(int rtArgc, void * rtArgv [])
{
    streamID = rtIOStreamOpen(rtArgc, rtArgv);
    if (streamID == RTIOSTREAM_ERROR)
    {
        return streamID;
    }
    return RTIOSTREAM_NO_ERROR;
}
```

Input Arguments

`argCount` — Argument count
scalar integer

Number of elements in `argValues` array.

`argValues` — Driver parameters
array of pointers to character vectors

Parameters for the communications driver.

Output Arguments

`streamID` — Stream handle
positive integer | -1

If the function initializes a communication stream, it returns a positive integer that represents the stream handle. Otherwise, it returns -1, which indicates an error.

The `rtiostream.h` file defines this macro:

```
#define RTIOSTREAM_ERROR (-1)
```

See Also

`rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose` | `rtiostream_wrapper`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation”
“Configure Processor-In-The-Loop (PIL) for a Custom Target”

Introduced in R2009a

rtIOStreamRecv

Receive data through communication channel

Syntax

```
errFlg = rtIOStreamRecv(streamID, dest, size, receivedDataSize)
```

Description

`errFlg = rtIOStreamRecv(streamID, dest, size, receivedDataSize)` receives data through a communication channel.

Examples

Send and Receive Data from Processor

This code from `rtiostreamtest.c` shows how to send and receive data from a target processor.

```
static void blockingIO(int send, unsigned long numMemUnits)
{
    size_t sizeToTransfer = (size_t) numMemUnits;
    size_t sizeTransferred;
    IOUnit_T * ioPtr = (IOUnit_T *) &buff[0];
    int status;

    if (numMemUnits > BUFFER_SIZE)
    {
        AckCode = stat_notEnoughSpace;
        AckArg0 = BUFFER_SIZE;
        return;
    }

#ifdef HOST_WORD_ADDRESSABLE_TESTING
   /* map to bytes */
   sizeToTransfer *= MEM_UNIT_BYTES;
#endif

   while (sizeToTransfer > 0) {
```

```
      sizeTransferred = 0;
      /* Do the low level call */
      status = send ?
         rtIOStreamSend(streamID, ioPtr, sizeToTransfer, &sizeTransferred) :
         rtIOStreamRecv(streamID, ioPtr, sizeToTransfer, &sizeTransferred);

      if (status != RTIOSTREAM_NO_ERROR) {
         if (AckCode == stat_OK) {
            AckCode = stat_RTIOSTREAM_ERROR;
            AckArg0 = data_counter;
         }
         return;
      }
      else {
         sizeToTransfer -= sizeTransferred;
         ioPtr += sizeTransferred;
      }
   }
}
```

Input Arguments

`streamID` — Stream handle

scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

`dest` — Data destination

void pointer

Pointer to the start of the buffer for received data.

`size` — Size of data to copy

size_t

Size of data to copy into the destination buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

`receivedDataSize` — Size of data received

size_t

Number of units of data received and copied into the buffer `dest`. If no data is copied, value is zero.

Output Arguments

`errFlg` — Error flag
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

See Also

`rtIOStreamSend` | `rtIOStreamOpen` | `rtIOStreamClose` | `rtiostream_wrapper`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation”
“Configure Processor-In-The-Loop (PIL) for a Custom Target”

Introduced in R2009a

rtIOStreamSend

Send data through communication channel

Syntax

```
errFlag = rtIOStreamSend(streamID, src, size, sizeSent)
```

Description

`errFlag = rtIOStreamSend(streamID, src, size, sizeSent)` sends data through a communication stream.

The API for `rtIOStream` functions is independent of the physical layer across which you send the data, for example, RS232, Ethernet, or Controller Area Network (CAN). The choice of physical layer affects the achievable data rates for communication between your development computer and target processor.

For a processor-in-the-loop (PIL) application, there is no minimum data rate requirement. The higher the data rate is, the faster the simulation runs.

A communications device driver can require additional hardware-specific or channel-specific configuration parameters. For example:

- A CAN channel can require the specification of the CAN node that is used.
- A TCP/IP channel can require the configuration of a port or static IP address.
- A CAN channel can require the specification of the CAN message ID and priority.

When you implement the `rtIOStream` driver functions, provide this configuration data, for example, by hard-coding the data or by supplying arguments to `rtIOStreamOpen`.

Examples

Send and Receive Data from Processor

This code from `rtiostreamtest.c` shows how to send and receive data from a target processor.

```
static void blockingIO(int send, unsigned long numMemUnits)
{
    size_t sizeToTransfer = (size_t) numMemUnits;
    size_t sizeTransferred;
    IOUnit_T * ioPtr = (IOUnit_T *) &buff[0];
    int status;

    if (numMemUnits > BUFFER_SIZE)
    {
        AckCode = stat_notEnoughSpace;
        AckArg0 = BUFFER_SIZE;
        return;
    }

#ifdef HOST_WORD_ADDRESSABLE_TESTING
   /* map to bytes */
   sizeToTransfer *= MEM_UNIT_BYTES;
#endif

   while (sizeToTransfer > 0) {
      sizeTransferred = 0;
      /* Do the low level call */
      status = send ?
         rtIOStreamSend(streamID, ioPtr, sizeToTransfer, &sizeTransferred) :
         rtIOStreamRecv(streamID, ioPtr, sizeToTransfer, &sizeTransferred);

      if (status != RTIOSTREAM_NO_ERROR) {
         if (AckCode == stat_OK) {
            AckCode = stat_RTIOSTREAM_ERROR;
            AckArg0 = data_counter;
         }
         return;
      }
      else {
         sizeToTransfer -= sizeTransferred;
         ioPtr += sizeTransferred;
      }
   }
}
```

Input Arguments

`streamID` — Stream handle
scalar integer

Handle to the stream returned by a previous call to `rtIOStreamOpen`.

`src` — Data source
constant void pointer

Pointer to the start of the buffer that contains a data array for transmission.

`size` — Size of data to transmit
size_t

Size of data to transmit from the source buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

Output Arguments

`errFlag` — Error flag
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

The `rtiostream.h` file defines these macros:

```
#define RTIOSTREAM_ERROR (-1)
#define RTIOSTREAM_NO_ERROR (0)
```

`sizeSent` — Size of transmitted data
size_t pointer

Size of transmitted data, which is less than or equal to `size`. If data is not transmitted, value is zero.

See Also

`rtIOStreamOpen` | `rtIOStreamClose` | `rtIOStreamRecv` | `rtiostream_wrapper`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation”
“Configure Processor-In-The-Loop (PIL) for a Custom Target”

Introduced in R2009a

rtiostream_wrapper

Test `rtiostream` shared library functions in MATLAB

Syntax

```
streamID = rtiostream_wrapper(sharedLib, 'open')
[errFlag,transmittedDataSize] = rtiostream_wrapper(sharedLib,'send',
streamID,data,dataSize)
[errFlag,receivedData,receivedDataSize] = rtiostream_wrapper(
sharedLib,'recv',streamID,dataSize)
streamID = rtiostream_wrapper( ___ ,Name,Value)
errFlag = rtiostream_wrapper(sharedLib,'close',streamID)
rtiostream_wrapper(sharedLib,'unloadlibrary')
```

Description

`streamID = rtiostream_wrapper(sharedLib, 'open')` opens an `rtiostream` communication channel or stream through a shared library.

`[errFlag,transmittedDataSize] = rtiostream_wrapper(sharedLib,'send', streamID,data,dataSize)`transmits data from a workspace variable through the open communication channel or stream.

`[errFlag,receivedData,receivedDataSize] = rtiostream_wrapper( sharedLib,'recv',streamID,dataSize)` receives workspace variable data from the open communication channel or stream.

`streamID = rtiostream_wrapper( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments. These arguments are implementation-dependent, that is, they are specific to the shared library that you use.

`errFlag = rtiostream_wrapper(sharedLib,'close',streamID)` closes the `rtiostream` communication channel or stream.

`rtiostream_wrapper(sharedLib,'unloadlibrary')` unloads the shared library, clearing persistent data.

Examples

Open Communication Channels

These examples use the supplied TCP/IP and serial communication drivers to open communication channels.

Open `rtiostream` `stationA` as a TCP/IP server:

```
stationA = rtiostream_wrapper('libmwrtiostreamtcpip.dll','open',...
                              '-client', '0',...
                              '-blocking', '0',...
                              '-port', port_number);
```

Opens `rtiostream` `StationB` as a TCP/IP client:

```
stationB = rtiostream_wrapper('libmwrtiostreamtcpip.dll','open',...
                              '-client','1',...
                              '-blocking', '0',...
                              '-port', port_number,...
                              '-hostname','localhost');
```

If you use the supplied development computer driver for serial communications (as an alternative to the drivers for TCP/IP), specify the bit rate when you open a channel with a specific port. For example, open channel `stationA` with port `COM1` and bit rate of 9600:

```
stationA = rtiostream_wrapper('libmwrtiostreamserial.dll','open',...
                              '-port','COM1',...
                              '-baud','9600');
```

Input Arguments

`sharedLib` — Shared library

character vector

Shared library that implements required `rtIOStream` functions, `rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv`, and `rtIOStreamClose`. Must be on system path. Specify one of these values:

- *`libTCPIP`* — For TCP/IP communication. Value depends on your operating system, for example, `'libmwrtiostreamtcpip.dll'`
- *`libSerial`* — For serial communication, for example, `'libmwrtiostreamserial.dll'`.

`streamID` — Stream handle
scalar integer

Handle to `rtiostream` communication stream.

`data` — Workspace variable
array

Array that contains data for transmission.

`dataSize` — Size of data to transmit or receive
scalar integer

Size of workspace variable data to transmit or receive, in bytes.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: '-hostname','localhost'

TCP/IP Communication

`-client` — Stream type
0 | 1

Open `rtiostream` as TCP/IP server or client:

- 0 — TCP/IP server
- 1 — TCP/IP client

`-port` — Port number
scalar integer

Port for TCP/IP communication.

`-hostname` — Development computer
character vector

Identifier for your development computer, for example, `'localhost'`.

`-blocking` — Call behavior

0 | 1

Call behavior when receiving data:

- 0 — Polling mode. If data is available, call returns with data. If data is not available, call returns without waiting.
- 1 — Blocking mode. If data is available, call returns with data. If data is not available, call waits for data. Use `recv_timeout_secs` to specify the waiting period.

Default is 0 unless the preprocessor macro `define VXWORKS` exists. In this case, the default is 1.

`-recv_timeout_secs` — Waiting period

positive integer | 0 | -1 | -2 | -3

Waiting period of call that receives data:

- *X*, a positive integer — Wait for *X* seconds.
- 0 — No waiting period.
- -1 — Wait indefinitely.
- -2 — Wait for default period.
- -3 — Wait 10 ms.

Default for client connections is to wait 1 second. Default for server connections is to wait indefinitely.

Serial Communication

`-port` — Port number

scalar integer

COM port for serial communication.

`-baud` — Bit rate

scalar integer

Bit rate for serial communication port.

Output Arguments

`streamID` — Stream handle
scalar integer

If the function opens the communications stream, it returns a handle to the stream. Otherwise, it returns -1.

`errFlag` — Error flag
0 | -1

If the function runs without errors, it returns zero. Otherwise, it returns -1.

`transmittedDataSize` — Size of transmitted data
scalar integer

Size of data transmitted through communication stream. Can be less than `dataSize`, the number of bytes specified for transmission.

`receivedData` — Workspace variable received
array

Array that contains received data.

`receivedDataSize` — Size of received data
scalar integer

Number of bytes received from communication stream. Can be less than `dataSize`, the number of bytes specified for transmission.

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Create a Target Communication Channel for Processor-In-The-Loop (PIL) Simulation”
“Configure Processor-In-The-Loop (PIL) for a Custom Target”

Introduced in R2008b

rtw.codegenObjectives.Objective class

Package: rtw.codegenObjectives

Customize code generation objectives

Description

An `rtw.codegenObjectives.Objective` object creates a code generation objective.

Construction

rtw.codegenObjectives.Objective	Create custom code generation objectives

Methods

addCheck	Add checks
addParam	Add parameters
excludeCheck	Exclude checks
modifyInheritedParam	Modify inherited parameter values
register	Register objective
removeInheritedCheck	Remove inherited checks
removeInheritedParam	Remove inherited parameters
setObjectiveName	Specify objective name

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

Examples

Create a custom objective named `Reduce RAM Example`. The following code is the contents of the `sl_customization.m` file that you create.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end

function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'DefaultParameterBehavior', 'Inlined');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'mathworks.design.UnconnectedLinesPorts');
addCheck(obj, 'mathworks.design.Update');

%Register the objective
register(obj);

end
```

See Also

Topics

“Create Custom Code Generation Objectives”

rtw.codegenObjectives.Objective

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Create custom code generation objectives

Syntax

```
obj = rtw.codegenObjectives.Objective('objID')
obj = rtw.codegenObjectives.Objective('objID', 'base_objID')
```

Description

`obj = rtw.codegenObjectives.Objective('objID')` creates an objective object, `obj`.

`obj = rtw.codegenObjectives.Objective('objID', 'base_objID')` creates an object, `obj`, for a new objective that is identical to an existing objective. You can then modify the new objective to meet your requirements.

Input Arguments

objID	A permanent, unique identifier for the objective. • You must have *objID*. • The value of *objID* must remain constant. • When you refresh your customizations, if *objID* is not unique, Simulink generates an error.
base_objID	The identifier of the objective that you want to base the new objective on.

Examples

Create a new objective:

```
obj = rtw.codegenObjectives.Objective('ex_ram_1');
```

Create a new objective based on the existing `Execution efficiency` objective:

```
obj = rtw.codegenObjectives.Objective('ex_my_efficiency_1', 'Execution efficiency');
```

See Also

Topics

“Create Custom Code Generation Objectives”

RTW.configSubsystemBuild

Package: RTW

Configure C function prototype or C++ class interface for right-click build of specified subsystem

Syntax

`RTW.configSubsystemBuild(`*`block`*`)`

Description

`RTW.configSubsystemBuild(`*`block`*`)` opens a graphical user interface where you can configure either C function prototype information or C++ class interface information for right-click builds of a specified nonvirtual subsystem. A dialog box opens based on the **Language** and **Code interface packaging** values selected for your model on the **Code Generation** and **Code Generation** > **Interface** panes of the Configuration Parameters dialog box.

To configure and generate C++ class interfaces for a nonvirtual subsystem, you must

- Select the system target file `ert.tlc` for the model.
- Select the **Language** parameter value `C++` for the model.
- Select the **Code interface packaging** parameter value `C++ class` for the model.
- Make sure that the subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

Input Arguments

`block`	Character vector specifying the name of a nonvirtual subsystem block in an ERT-based Simulink model.

See Also

Topics

“Customize Function Interfaces for Nonvirtual Subsystems”
“Customize Generated C Function Interfaces”
“Configure C++ Class Interfaces for Nonvirtual Subsystems”
“Customize Generated C++ Class Interfaces”

Introduced in R2008b

rtw.connectivity.ComponentArgs

Provide parameters for each target connectivity component

Description

An `rtw.connectivity.ComponentArgs` object provides functions for getting information about the source component and the target application.

Creation

Description

`compArgs = rtw.connectivity.ComponentArgs (componentPath, componentCodePath, componentCodeName, applicationCodePath)` returns a handle to an `rtw.connectivity.ComponentArgs` object.

Object Functions

Function	Description
`getComponentPath`	*`cmpPath = compArgs`*`.getComponentPath` returns the system path of the source component. For example: • For MATLAB, the path of the function that is under test. • For Simulink, the path of the referenced model that is under test.

Function	Description
`getComponentCodePath`	*`cmpCodePath`* `= `*`compArgs`*`.getComponentCodePath` returns the code generation folder path associated with the source component. For example: • For MATLAB, the code generation folder of the MATLAB function that is under test. • For Simulink, the code generation folder of the referenced model that is under test.
`getComponentCodeName`	*`cmpCodeName`* `= `*`compArgs`*`.getComponentCodeName` returns the component name used for code generation.
`getApplicationCodePath`	*`appCodePath`* `= `*`compArgs`*`.getApplicationCodePath` returns the folder path associated with the target application, for example, the path associated with the PIL application.
`getParam`	*`settingOrParameterValue`* `= `*`compArgs`*`.getParam(`*`settingOrParameterName`*`)` returns: • For MATLAB, the value of the specific MATLAB Coder setting for the generated code. • For Simulink, the value of the specific model configuration parameter for the generated code. The function does not load the model.

Examples

Using `rtw.connectivity.ComponentArgs` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.Config`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

Introduced in R2008b

rtw.connectivity.Config

Define connectivity implementation that comprises builder, launcher, and communicator components

Description

The `rtw.connectivity.Config` class specifies the actions required for running a processor-in-the-loop (PIL) simulation.

Creation

Description

`rtw.connectivity.Config(componentArgs, builder, launcher, communicator)` creates an `rtw.connectivity.Config` object with these arguments:

- *`componentArgs`* – `rtw.connectivity.ComponentArgs` object
- *`builder`* – `rtw.connectivity.Builder` object, for example, `rtw.connectivity.MakefileBuilder` object.
- *`launcher`* – `rtw.connectivity.Launcher` object
- *`communicator`* – `rtw.connectivity.Communicator`, for example, `rtw.connectivity.RtIOStreamHostCommunicator` object.

To define a connectivity implementation:

1. Create a subclass of `rtw.connectivity.Config` that creates instances of your connectivity component classes:
   - `rtw.connectivity.MakefileBuilder`
   - `rtw.connectivity.Launcher`
   - `rtw.connectivity.RtIOStreamHostCommunicator`
2. Define the constructor for your subclass:

```
function this = myConfig(componentArgs)
```

When the software creates an instance of your subclass of `rtw.connectivity.Config`, it provides an instance of the `rtw.connectivity.ComponentArgs` class as the only constructor argument. If you want to test your subclass of `rtw.connectivity.Config` manually, you can create an `rtw.connectivity.ComponentArgs` object to pass as a constructor argument.

3. After instantiating the builder, launcher and communicator objects in your subclass, call the constructor of the superclass `rtw.connectivity.Config` to define your complete target connectivity configuration. For example:

```
this@rtw.connectivity.Config(componentArgs,...
builder, launcher, communicator);
```

4. Optionally, for execution-time profiling, use the `setTimer` method to register your hardware timer. For example, if you specified the timer in a code replacement table, insert the following line:

```
this.setTimer('myCrlTable')
```

`myCrlTable` is the name of the code replacement table, which must be in a location on the MATLAB search path.

You can also estimate and remove instrumentation overheads from execution-time measurements. For example:

```
this.activateOverheadFiltering(true);
this.runOverheadBenchmark(true);
this.setOverheadBenchmarkSteps(50);
```

Register your subclass name, for example, `myPIL.ConnectivityConfig` by using the class `rtw.connectivity.ConfigRegistry`. The PIL infrastructure instantiates your subclass as required. The `rtwTargetInfo.m` file (for MATLAB) or `sl_customization.m` mechanism (for Simulink) specifies a suitable connectivity configuration for use with a particular PIL component (and its configuration set). The subclass can also perform additional validation on construction. For example, you can use the component path returned by the `getComponentPath` method of the `componentArgs` constructor argument to query and validate parameters associated with the PIL component under test.

Examples

Using `rtw.connectivity.Config` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.MakefileBuilder` | `rtw.connectivity.Launcher` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.connectivity.ComponentArgs`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Specify Hardware Timer”
“Specify Hardware Timer”
“Remove Instrumentation Overheads from Execution Time Measurements”
“SIL and PIL Limitations”

Introduced in R2008b

rtw.connectivity.ConfigRegistry

Register connectivity configuration

Description

Register your connectivity configuration with MATLAB or Simulink.

Creation

Description

`config = rtw.connectivity.ConfigRegistry` returns a handle to an `rtw.connectivity.ConfigRegistry` object.

To create this class:

- For MATLAB, use an `rtwTargetInfo.m` file, which you must place on the MATLAB search path. In the `rtwTargetInfo.m` file, a call to `registerTargetInfo` registers the connectivity configuration.
- For Simulink, use an `sl_customization.m` file, which you must place on the MATLAB search path. When Simulink starts, it reads the file, and registers your connectivity configuration through a call to `registerTargetInfo` in the file.

Through the first two properties of this class, you can specify for your connectivity configuration:

- A unique name.
- An associated connectivity implementation class, which is a subclass of `rtw.connectivity.Config`.

Through the remaining properties, you can define:

- For MATLAB, the code that is compatible with the connectivity implementation class.
- For Simulink, the set of models that are compatible with the connectivity implementation class.

A comparison of the union of these properties against the MATLAB Coder configuration settings or Simulink model parameters determines compatibility. For example with Simulink, whether the `SystemTargetFile`, `TemplateMakefile`, and `HardwareBoard` properties jointly match the corresponding model parameters.

Properties

`ConfigName` — Name

character vector

Unique name for configuration.

`ConfigClass` — Class name

character vector

Full class name of the connectivity implementation that you want to register.

`SystemTargetFile` — System target files (Simulink)

cell array of character vectors | {}

For Simulink, system target files that support the `rtw.connectivity.ConfigRegistry` object you create. A comparison of this cell array against the `SystemTargetFile` configuration parameter of the model determines whether the created object is valid for use. An empty cell array matches any system target file.

`TemplateMakefile` — Template makefiles (Simulink)

cell array of character vectors | {}

For Simulink, template makefiles that support the `rtw.connectivity.ConfigRegistry` object you create. A comparison of this cell array against the `TemplateMakefile` configuration parameter of the model determines whether the created object is valid for use. An empty cell array matches any template makefile and non-makefile target (`GenerateMakefile`: off).

If you use a toolchain to build the generated code, do not specify the `TemplateMakefile` configuration parameter. Instead, specify the `Toolchain` configuration parameter.

`Toolchain` — Toolchains

cell array of character vectors | {}

Toolchains that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `Toolchain` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `Toolchain` configuration parameter of the model determines whether the created object is valid for use. If you do not use a toolchain to build the generated code, do not specify the `Toolchain` configuration parameter. Instead, specify the `TemplateMakefile` configuration parameter.

An empty cell array matches any toolchain.

`HardwareBoard` — Hardware boards
cell array of character vectors | {}

Hardware boards that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `HardwareBoard` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `HardwareBoard` configuration parameter of the model determines whether the created object is valid for use.

An empty cell array matches any hardware board.

`TargetHWDeviceType` — Hardware device types
cell array of character vectors | {}

Hardware device types that support the `rtw.connectivity.ConfigRegistry` object you create:

- For MATLAB, a comparison of this cell array against the MATLAB Coder `TargetHWDeviceType` configuration setting determines whether the created object is valid for use.
- For Simulink, a comparison of this cell array against the `TargetHWDeviceType` configuration parameter of the model determines whether the created object is valid for use.

An empty cell array matches any hardware device type.

Examples

Create `rtwTargetInfo.m` File

This code is an example `rtwTargetInfo.m` file. Use the function syntax exactly as shown.

```
function rtwTargetInfo(tr)
% Register PIL connectivity config: mypil.ConnectivityConfig

tr.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

% Create object for connectivity configuration
config = rtw.connectivity.ConfigRegistry;
% Assign connectivity configuration name
config.ConfigName = 'My PIL Example';
% Associate the connectivity configuration with the connectivity
% API implementation
config.ConfigClass = 'mypil.ConnectivityConfig';

% Specify toolchains for host-based PIL
config.Toolchain =  rtw.connectivity.Utils.getHostToolchainNames;

% Through the HardwareBoard and TargetHWDeviceType properties,
% define compatible code for the target connectivity configuration
config.HardwareBoard = {};
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

The function performs the following steps:

1. Creates an instance of the `rtw.connectivity.ConfigRegistry` class. For example:

   ```
   config = rtw.connectivity.ConfigRegistry;
   ```

2. Assigns a connectivity configuration name to the `ConfigName` property of the object. For example:

```
config.ConfigName = 'My PIL Example';
```

3. Associates the connectivity configuration with the connectivity API implementation created in step 1. For example:

   ```
   config.ConfigClass = 'mypil.ConnectivityConfig';
   ```

4. Defines compatible code for this target connectivity configuration, by setting the `HardwareBoard` and `TargetHWDeviceType` properties of the object. For example:

   ```
   config.HardwareBoard = {};  % Any hardware board
   config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                                'Generic->Custom' ...
                                'Intel->x86-64 (Windows64)', ...
                                'Intel->x86-64 (Mac OS X)', ...
                                'Intel->x86-64 (Linux 64)'};
   ```

Create `sl_customization.m` File

This code is an example of an `sl_customization.m` file. Use the `sl_customization.m` file structure, and call the `registerTargetInfo` function exactly as shown.

```
function sl_customization(cm)
% SL_CUSTOMIZATION for PIL connectivity config:...
% mypil.ConnectivityConfig

% Copyright 2008 The MathWorks, Inc.

cm.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

config = rtw.connectivity.ConfigRegistry;
config.ConfigName = 'My PIL Example';
config.ConfigClass = 'mypil.ConnectivityConfig';

% Match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};

% If you use a toolchain to build your generated code,
% specify the config.Toolchain property to match your
% Simulink model toolchain setting. Otherwise, for a
% non-toolchain approach, match the TMF
```

```
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};

% Match hardware boards and hardware device types
config.HardwareBoard = {};  % Any hardware board
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'Intel->x86-64 (Windows64)', ...
                             'Intel->x86-64 (Mac OS X)', ...
                             'Intel->x86-64 (Linux 64)'};
```

You must configure the file to perform the following steps when Simulink starts:

1. Create an instance of the `rtw.connectivity.ConfigRegistry` class. For example:

   ```
   config = rtw.connectivity.ConfigRegistry;
   ```

2. Assign a connectivity configuration name to the `ConfigName` property of the object. For example:

   ```
   config.ConfigName = 'My PIL Example';
   ```

3. Associate the connectivity configuration with the connectivity API implementation (created in step 1). For example:

   ```
   config.ConfigClass = 'mypil.ConnectivityConfig';
   ```

4. Define compatible models for this target connectivity configuration, by setting these properties of the properties of the object:

   - `SystemTargetFile`
   - `Toolchain` or `TemplateMakefile`
   - `HardwareBoard`
   - `TargetHWDeviceType`

   For example:

   ```
   config.SystemTargetFile = {'ert.tlc'};
   config.TemplateMakefile = {'ert_default_tmf' ...
                              'ert_unix.tmf', ...
                              'ert_vcx64.tmf', ...
   ```

```
                              'ert_lcc.tmf'};
config.HardwareBoard = {};
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                              'Generic->Custom' ...
                              'Intel->x86-64 (Windows64)', ...
                              'Intel->x86-64 (Mac OS X)', ...
                              'Intel->x86-64 (Linux 64)'};
```

Using `rtw.connectivity.ConfigRegistry` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.Config`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Register Code Replacement Mappings”

Introduced in R2008b

rtw.connectivity.Launcher

Control downloading, starting, and resetting of a target application

Description

The `rtw.connectivity.Launcher` class, which runs on your development computer, controls execution of an application on the target processor.

Creation

Description

`rtw.connectivity.Launcher(componentArgs)` controls the download, start, and reset of an application, for example, a PIL application.

Make a subclass and implement the `startApplication` and `stopApplication` methods.

You can implement a destructor method that cleans up resources (for example, a handle to a third-party download tool) when the object is cleared from memory.

Object Functions

Function	Description
`getComponentArgs`	`componentArgs = obj.getComponentArgs` returns the `rtw.connectivity.ComponentArgs` object associated with the launcher object.
`setExe`	`setExe(exe)` specifies the application that runs on the target processor.
`getExe`	`exe=getExe()` returns the application that is running on the target processor.

Function	Description
`startApplication`	`obj.startApplication` is an abstract method that you implement in a subclass. Called by MATLAB or Simulink to start execution of the target application. MATLAB or Simulink calls the `setExe` method, which specifies the target application to run. To obtain this application, use the `getExe` method. For example: `exe = getExe()` The `startApplication` method resets the application to its initial state by ensuring that external and static (global) variables are zero-initialized.
`stopApplication`	`obj.stopApplication` is an abstract method that you must implement in a subclass. Called by MATLAB to stop execution of the target application.
`getBuilder`	`builder = obj.getBuilder` returns the `rtw.connectivity.Builder` object associated with the launcher object.

Examples

Using `rtw.connectivity.Launcher` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.MakefileBuilder` |
`rtw.connectivity.RtIOStreamHostCommunicator` |
`rtw.connectivity.ComponentArgs`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

Introduced in R2008b

rtw.connectivity.MakefileBuilder

Configure toolchain-based build process

Description

Control toolchain-based build process for the creation of a PIL application.

Creation

Description

`rtw.connectivity.MakefileBuilder(componentArgs, targetApplicationFramework, exeExtension)` creates an object with these arguments:

- *`componentArgs`* - An `rtw.connectivity.ComponentArgs` object
- *`TargetApplicationFramework`* - An `rtw.pil.RtIOStreamApplicationFramework` object. For example, `myPIL.TargetFramework`.
- *`exeExtension`* - Name extension of executable file for target system. The extension depends on the toolchain defined by `rtw.connectivity.ConfigRegistry`. For an embedded target, the extension can be, for example, `'.elf'`, `'.abs'`, `'.sre'`, or `'.hex'`. For a Windows development computer target, the extension is `'.exe'`. For a UNIX® development computer target, the extension is empty, `''`.

If you use the template makefile approach to build the PIL application, you must provide a template makefile that includes these tokens:

- `MAKEFILEBUILDER_TGT`
- `STANDALONE_SUPPRESS_EXE`

You can create the template makefile by customizing a copy of one of the supplied ERT template makefiles, for example, `ert_unix.tmf` or `ert_vc.tmf`. You must associate the `MAKEFILEBUILDER_TGT` and `STANDALONE_SUPPRESS_EXE` tokens with corresponding makefile rules. For more information, see “Customize Template Makefiles”.

Examples

Using `rtw.connectivity.MakefileBuilder` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtw.connectivity.ComponentArgs` |
`rtw.pil.RtIOStreamApplicationFramework`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

Introduced in R2008b

rtw.connectivity.RtIOStreamHostCommunicator

Configure development computer communications with target processor

Description

Configure communications between your development computer and the target processor by loading and initializing a shared library that implements the `rtiostream` functions.

Creation

Description

`rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, launcher, rtiostreamLib)` creates an object by using these arguments:

- *`componentArgs`* –– `rtw.connectivity.ComponentArgs` object.
- *`launcher`* –– `rtw.connectivity.Launcher` object.
- *`rtiostreamLib`* –– `rtiostream` shared library that implements the development computer part of communications between the development computer and the target processor.

The object loads and initializes the shared library.

For your development computer, Embedded Coder provides a shared library for these communication protocols:

- TCP/IP
- serial

You must provide drivers for the target processors.

For other communication protocols, for example, USB, you must provide a shared library for the development computer and drivers for the target processors.

To create your instance of `rtw.connectivity.RtIOStreamHostCommunicator`, you have these options:

- Instantiate `rtw.connectivity.RtIOStreamHostCommunicator` directly, providing custom arguments for the `rtiostream` shared library.
- Create a subclass of `rtw.connectivity.RtIOStreamHostCommunicator`. Consider this option when more complex configuration is required. For example, when:
  - The subclass `rtw.connectivity.HostTCPIPCommunicator` includes additional code to determine the number of the TCP/IP port that the executable application serves.
  - You use a subclass to specify a serial port number.
  - You specify verbose or silent operation.

Object Functions

Function	Description
`setTimeoutRecvSecs`	`hostCommunicator.setTimeoutRecvSecs(`*`timeout`*`)` sets the timeout value for reading data. You can configure data reading to time out if no new data is received for a period greater than `timeout` seconds.
`setInitCommsTimeout`	`hostCommunicator.setInitCommsTimeout(`*`timeout`*`)` sets the timeout value for initial setup of the communications channel. For some target processors, you might need to set a timeout value for initial setup of the communications channel. For example, the target processor can take a few seconds to open its side of the communications channel. If you set a nonzero timeout value, the communicator repeatedly tries to open the communications channel until the timeout value is reached.

Examples

Using `rtw.connectivity.RtIOStreamHostCommunicator` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtiostream_wrapper` | `rtw.connectivity.ComponentArgs` | `rtw.connectivity.Launcher`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”

Introduced in R2008b

RTW.getClassInterfaceSpecification

Package: RTW

Get handle to model-specific C++ class interface control object

Syntax

```
obj = RTW.getClassInterfaceSpecification(modelName)
```

Description

`obj = RTW.getClassInterfaceSpecification(modelName)` returns a handle to a model-specific C++ class interface control object.

Input Arguments

modelName	Character vector specifying the name of a loaded ERT-based Simulink model.

Output Arguments

obj	Handle to the C++ class interface control object associated with the specified model. If the model does not have an associated C++ class interface control object, the function returns `[]`.

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C

++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

Introduced in R2014a

RTW.getFunctionSpecification

Package: RTW

Get handle to model-specific C prototype function control object

Syntax

```
obj = RTW.getFunctionSpecification(modelName)
```

Description

`obj = RTW.getFunctionSpecification(modelName)` returns a handle to the model-specific C function prototype control object.

Input Arguments

modelName	Character vector specifying the name of a loaded ERT-based Simulink model.

Output Arguments

obj	Handle to the model-specific C prototype function control object associated with the specified model. If the model does not have an associated function control object, the function returns `[]`.

Alternatives

You can use the Configure C Step Function Interface dialog box to customize the base rate C step function for a rate-based model. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

Introduced in R2008a

RTW.ModelCPPArgsClass class

Package: RTW
Superclasses:

Control C++ class interfaces for models using I/O arguments style step method

Description

The `ModelCPPArgsClass` class provides objects that describe C++ class interfaces for models using an I/O arguments style step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

Construction

RTW.ModelCPPArgsClass	Create C++ class interface object for configuring model class with I/O arguments style step method

Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following methods.

getArgCategory	Get argument category for Simulink model port from model-specific C++ class interface
getArgName	Get argument name for Simulink model port from model-specific C++ class interface
getArgPosition	Get argument position for Simulink model port from model-specific C++ class interface
getArgQualifier	Get argument type qualifier for Simulink model port from model-specific C++ class interface
runValidation	Validate model-specific C++ class interface against Simulink model
setArgCategory	Set argument category for Simulink model port in model-specific C++ class interface
setArgName	Set argument name for Simulink model port in model-specific C++ class interface
setArgPosition	Set argument position for Simulink model port in model-specific C++ class interface
setArgQualifier	Set argument type qualifier for Simulink model inport in model-specific C++ class interface

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB).

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

RTW.ModelCPPArgsClass

Class: RTW.ModelCPPArgsClass
Package: RTW

Create C++ class interface object for configuring model class with I/O arguments style step method

Syntax

```
obj = RTW.ModelCPPArgsClass
```

Description

`obj = RTW.ModelCPPArgsClass` returns a handle, *obj*, to a newly created object of class `RTW.ModelCPPArgsClass`.

Output Arguments

obj	Handle to a newly created C++ class interface object for configuring a model class with an I/O arguments style step method. The object has not yet been configured or attached to an ERT-based Simulink model.

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

RTW.ModelCPPClass class

Package: RTW

Control C++ class interfaces for models

Description

The `ModelCPPClass` class is the base class for the classes `RTW.ModelCPPArgsClass` and `RTW.ModelCPPDefaultClass`, which provide objects that describe C++ class interfaces for models using either an I/O arguments style step method or a default style step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

Construction

To access the methods of this class, use the constructor for either `RTW.ModelCPPArgsClass` or `RTW.ModelCPPDefaultClass`.

Methods

attachToModel	Attach model-specific C++ class interface to loaded ERT-based Simulink model
getClassName	Get class name from model-specific C++ class interface
getDefaultConf	Get default configuration information for model-specific C++ class interface from Simulink model
getNamespace	Get namespace from model-specific C++ class interface
getNumArgs	Get number of step method arguments from model-specific C++ class interface
getStepMethodName	Get step method name from model-specific C++ class interface
setClassName	Set class name in model-specific C++ class interface
setNamespace	Set namespace in model-specific C++ class interface
setStepMethodName	Set step method name in model-specific C++ class interface

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C ++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

RTW.ModelCPPDefaultClass class

Package: RTW
Superclasses:

Control C++ class interfaces for models using default model step method

Description

The `ModelCPPDefaultClass` class provides objects that describe C++ class interfaces for models using a default model step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

Construction

RTW.ModelCPPDefaultClass	Create C++ class interface object for configuring model class with default model step method

Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following method.

runValidation	Validate model-specific C++ class interface against Simulink model

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB).

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog

box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C ++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

RTW.ModelCPPDefaultClass

Class: RTW.ModelCPPDefaultClass
Package: RTW

Create C++ class interface object for configuring model class with default model step method

Syntax

```
obj = RTW.ModelCPPDefaultClass
```

Description

`obj = RTW.ModelCPPDefaultClass` returns a handle, *obj*, to a newly created object of class `RTW.ModelCPPDefaultClass`.

Output Arguments

obj	Handle to a newly created C++ class interface object for configuring a model class with a default model step method. The object has not yet been configured or attached to an ERT-based Simulink model.

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. See “Customize C++ Class Interfaces Using Graphical Interfaces”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

RTW.ModelSpecificCPrototype class

Package: RTW

Describe signatures of functions for model

Description

A `ModelSpecificCPrototype` object describes the signatures of the step and initialization functions for a model. You must use this in conjunction with the `attachToModel` method.

Construction

RTW.ModelSpecificCPrototype	Create model-specific C prototype object

Methods

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
getFunctionName	Get function name from model-specific C function prototype
getNumArgs	Get number of function arguments from model-specific C function prototype
getPreview	Get model-specific C function prototype code preview
runValidation	Validate model-specific C function prototype against Simulink model
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 inport in model-specific C function prototype
setFunctionName	Set function name in model-specific C function prototype

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB).

Examples

The code below creates a function control object, `a`, and uses it to add argument configuration information to the model.

```
% Open the rtwdemo_counter model and specify the System Target File
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)
```

Alternatives

You can create a function control object using the Model Interface dialog box.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

Topics

“Customize Generated C Function Interfaces”

RTW.ModelSpecificCPrototype

Class: RTW.ModelSpecificCPrototype
Package: RTW

Create model-specific C prototype object

Syntax

```
obj = RTW.ModelSpecificCPrototype
```

Description

`obj = RTW.ModelSpecificCPrototype` creates a handle, *obj*, to an object of class `RTW.ModelSpecificCPrototype`.

Output Arguments

obj	Handle to model specific C prototype object.

Examples

Create a function control object, `a`, and use it to add argument configuration information to the model:

```
% Open the rtwdemo_counter model and specify the System Target File
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)
```

Alternatives

Use the Configure C Step Function Interface dialog box to customize the base rate C step function for a rate-based model. See “Override Default C Step Function Interface”.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

Topics

“Customize Generated C Function Interfaces”

rtw.pil.RtIOStreamApplicationFramework

Configure target-side communications

Description

Specify target-specific libraries and source files that are required to build the executable file. The libraries and source files must include the device drivers that implement the target-side of the `rtiostream` communications channel.

Creation

Description

`appFrameObj = rtw.pil.RtIOStreamApplicationFramework(componentArgs)` returns an object that provides access to an `RTW.BuildInfo` object containing PIL-specific files (including a PIL `main` function). `rtw.connectivity.MakefileBuilder` combines these files with the PIL component libraries to create the PIL application.

Make a subclass of `rtw.pil.RtIOStreamApplicationFramework`. In addition:

- Use the `addPILMain` method to specify a `main` function, which is required to build the PIL application.
- To the `RTW.BuildInfo` object, add data that is required for the implementation of the `rtiostream` target communications interface by using provided functions:
  - Source file names - `addSourceFiles`
  - Source file paths - `addSourcePaths`
  - Include file names - `addIncludeFiles`
  - Include file paths - `addIncludePaths`
  - Libraries - `addLinkObjects`
  - Preprocessor macro definitions - `addDefines`
  - Compiler options - `addCompileFlags`

- Linker options - `addLinkFlags`

Object Functions

Function	Description
`getComponentArgs`	`componentArgs = appFrameObj.getComponentArgs` returns the `rtw.connectivity.ComponentArgs` object associated with `appFrameObj`.
`getBuildInfo`	`buildInfo = appFrameObj.getBuildInfo` returns the `RTW.BuildInfo` object associated with `appFrameObj`.

`addPILMain`	To build the PIL application, a `main` function is required. Use this method to add one of the two provided files to the application framework.
	
	To specify a `main` function that is adapted for on-target PIL and suitable for most PIL implementations, enter:
	
	`appFrameObj.addPILMain('target');`
	
	To specify a `main` function that is adapted for PIL on your development computer, enter:
	
	`appFrameObj.addPILMain('host');`
	
	Alternatively, you can specify your own `main` function:
	
	`componentArgs = appFrameObj.getComponentArgs;`
	`buildInfo = appFrameObj.getBuildInfo;`
	`buildInfo.addSourcePaths(`*`pathToMyMainC`*`);`
	`buildInfo.addSourceFiles(`*`myMainC`*`);`

Examples

Using `rtw.pil.RtIOStreamApplicationFramework` in PIL Connectivity

For an example that shows how to use this object in setting up PIL connectivity, see:

- For MATLAB, “Processor-in-the-Loop Execution From Command Line”
- For Simulink, “Configure Processor-In-The-Loop (PIL) for a Custom Target”

See Also

`rtiostream_wrapper` | `rtw.connectivity.ComponentArgs`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”
“Create PIL Target Connectivity Configuration for Simulink”
“Build Information Object” (Simulink Coder)

Introduced in R2008b

run

Class: cgv.CGV
Package: cgv

Execute CGV object

Syntax

```
result = cgvObj.run()
```

Description

`result = cgvObj.run()` executes the model once for each input data that you added to the object. `result` is a boolean value that indicates whether the run completed without execution error. `cgvObj` is a handle to a `cgv.CGV` object.

After each execution of the model, the object captures and writes the following metadata to a file in the output folder:
`ErrorDetails` — If errors occur, the error information.
`status` — The execution status.
`ver` — Version information for MathWorks® products.
`hostname` — Name of computer.
`dateTime` — Date and time of execution.
`warnings` — If warnings occur, the warning messages.
`username` — Name of user.
`runtime` — The amount of time that lapsed for the execution.

Tips

- Only call `run` once for each `cgv.CGV` object.
- The cgv.CGV methods that set up the object are ignored after a call to `run`. See the `cgv.CGV` for details.
- You can call `run` once without first calling `addInputData`. However, it is recommended that you first save the required data for execution to a MAT-file,

including the model inputs and parameters. Then use `cgv.CGV.addInputData` to pass the MAT-file to the CGV object before calling `run`.

- The `cgv.CGV` object supports callback functions that you can define and add to the `cgv.CGV` object. These callback functions are called during `cgv.CGV.run()` in the following order:

Callback function	Add to object using...	`cgv.CGV.run()` executes callback function...
HeaderReportFcn	`addHeaderReportFcn`	Before executing input data in `cgv.CGV`
PreExecReportFcn	`addPreExecReportFcn`	Before executing each input data file in `cgv.CGV`
PreExecFcn	`addPreExecFcn`	Before executing each input data file in `cgv.CGV`
PostExecReportFcn	`addPostExecReportFcn`	After executing each input data file in `cgv.CGV`
PostExecFcn	`addPostExecFcn`	After executing each input data file in `cgv.CGV`
TrailerReportFcn	`addTrailerReportFcn`	After the input data is executed in `cgv.CGV`

See Also

Topics

“Verify Numerical Equivalence with CGV”

runValidation

Class: RTW.ModelCPPArgsClass
Package: RTW

Validate model-specific C++ class interface against Simulink model

Syntax

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

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ class interface 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.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = RTW.ModelCPPArgsClass on page 1-412 or *obj* = RTW.getClassInterfaceSpecification (*modelName*).

Output Arguments

status	Boolean value; true for a valid configuration, false otherwise.

msg	If *status* is false, *msg* contains a character vector of information describing why the configuration is invalid.

Alternatives

To validate a C++ class interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays status and an explanation of failures. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

runValidation

Class: RTW.ModelCPPDefaultClass
Package: RTW

Validate model-specific C++ class interface against Simulink model

Syntax

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

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ class interface 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.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPDefaultClass` on page 1-418 or *obj* = `RTW.getClassInterfaceSpecification (`*modelName*`)`.

Output Arguments

status	Boolean value; true for a valid configuration, false otherwise.

msg	If *status* is false, *msg* contains a character vector of information describing why the configuration is invalid.

Alternatives

To validate a C++ class interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays status and an explanation of failures. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

runValidation

Class: RTW.ModelSpecificCPrototype
Package: RTW

Validate model-specific C function prototype against Simulink model

Syntax

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

Description

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

Input Arguments

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

Output Arguments

status	True for a valid configuration; false otherwise.
msg	If *status* is false, *msg* contains a character vector explaining why the configuration is invalid.

Alternatives

Click the **Validate** button on the Configure C Step Function Interface dialog box to run a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

setArgCategory

Class: RTW.ModelCPPArgsClass
Package: RTW

Set argument category for Simulink model port in model-specific C++ class interface

Syntax

`setArgCategory(obj, portName, category)`

Description

`setArgCategory(obj, portName, category)` sets the category — `'Value'`, `'Pointer'`, or `'Reference'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-412 or `obj = RTW.getClassInterfaceSpecification (modelName)`.
portName	Character vector specifying the unqualified name of an inport or outport in your Simulink model.
category	Character vector specifying the argument category — `'Value'`, `'Pointer'`, or `'Reference'` — 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 causes the argument to move to the first argument position when `attachToModel` or `runValidation` is called.

Alternatives

To set argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument categories that you can examine and modify. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

setArgCategory

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

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

Description

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

Input Arguments

`obj`	Handle to a model-specific C prototype function control object previously returned by *`obj`* `= RTW.ModelSpecificCPrototype` or *`obj`* `= RTW.getFunctionSpecification(`*`modelName`*`)`.
`portName`	Character vector specifying the unqualified name of an inport or outport in your Simulink model.
`category`	Character vector specifying the argument category, `'Value'` or `'Pointer'`, that you set for the specified Simulink model port.

Note If you change the argument category for an outport from `'Pointer'` to `'Value'`, it causes the argument to move to the first argument position when you call `RTW.ModelSpecificCPrototype.attachToModel` or `RTW.ModelSpecificCPrototype.runValidation`.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

setArgName

Class: RTW.ModelCPPArgsClass
Package: RTW

Set argument name for Simulink model port in model-specific C++ class interface

Syntax

`setArgName(obj, portName, argName)`

Description

`setArgName(obj, portName, argName)` sets the argument name that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = RTW.ModelCPPArgsClass on page 1-412 or *obj* = RTW.getClassInterfaceSpecification (*modelName*).
portName	Character vector specifying the name of an inport or outport in your Simulink model.
argName	Character vector specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

To set argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument names that you can examine and modify. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

setArgName

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

`setArgName(obj, portName, argName)`

Description

`setArgName(obj, portName, argName)` sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport or outport in your Simulink model.
argName	Character vector specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

setArgPosition

Class: RTW.ModelCPPArgsClass
Package: RTW

Set argument position for Simulink model port in model-specific C++ class interface

Syntax

`setArgPosition(obj, portName, position)`

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ class interface. The specified argument is then moved to the specified position, and other arguments shifted by one position accordingly.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` on page 1-412 or `obj = RTW.getClassInterfaceSpecification (modelName)`.
portName	Character vector 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.

Alternatives

To set argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument positions that you can examine and modify. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

setArgPosition

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

`setArgPosition(obj, portName, position)`

Description

`setArgPosition(obj, portName, position)` 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 moves to the specified position, and other arguments shift by one position accordingly.

Input Arguments

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

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

setArgQualifier

Class: RTW.ModelCPPArgsClass
Package: RTW

Set argument type qualifier for Simulink model inport in model-specific C++ class interface

Syntax

`setArgQualifier(obj, portName, qualifier)`

Description

`setArgQualifier(obj, portName, qualifier)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument that corresponds to a specified Simulink model inport in a specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = RTW.ModelCPPArgsClass on page 1-412 or *obj* = RTW.getClassInterfaceSpecification (*modelName*).
portName	Character vector specifying the name of an inport in your Simulink model.
qualifier	Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — to be set for the specified Simulink model inport.

Note If you specify a qualifier for an outport, the code generator ignores the argument setting.

Alternatives

To set argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers that you can examine and modify. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

setArgQualifier

Class: RTW.ModelSpecificCPrototype
Package: RTW

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

Syntax

`setArgQualifier(obj, portName, qualifier)`

Description

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

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`.
portName	Character vector specifying the name of an inport in your model.
qualifier	Character vector specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — to be set for the specified model inport.

Note If you specify a qualifier for an outport, the code generator ignores the argument setting.

Alternatives

Use the Configure C Step Function Interface dialog box to configure C step function arguments. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

setClassName

Class: RTW.ModelCPPClass
Package: RTW

Set class name in model-specific C++ class interface

Syntax

`setClassName(obj, clsName)`

Description

`setClassName(obj, clsName)` sets the class name in the specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = RTW.ModelCPPArgsClass on page 1-412, *obj* = RTW.ModelCPPDefaultClass on page 1-418, or *obj* = RTW.getClassInterfaceSpecification (*modelName*).
clsName	Character vector specifying a new name for the class described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments`

`step method` view of this dialog box, click the **Get Default Configuration** button to display the model class name, which you can examine and modify. In the `Default step method` view, you can examine and modify the model class name without having to click a button. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

setFunctionName

Class: RTW.ModelSpecificCPrototype
Package: RTW

Set function name in model-specific C function prototype

Syntax

`setFunctionName(`*`obj`*`, `*`fcnName`*`, `*`fcnType`*`)`

Description

`setFunctionName(`*`obj`*`, `*`fcnName`*`, `*`fcnType`*`)` sets the step or initialization function name in the specified function control object.

Input Arguments

`obj`	Handle to a model-specific C prototype function control object previously returned by *`obj`* `= RTW.ModelSpecificCPrototype` or *`obj`* `= RTW.getFunctionSpecification(`*`modelName`*`)`.
`fcnName`	Character vector specifying a new name for the function described by the function control object. The argument must be a valid C identifier.
`fcnType`	Optional. Character vector specifying which function to name. Valid options are `'step'` and `'init'`. If *`fcnType`* is not specified, sets the step function name.

Alternatives

Use the **Step function name** field on the Configure C Step Function Interface dialog box to configure the C step function name. See “Override Default C Step Function Interface”.

See Also

Topics

“Customize Generated C Function Interfaces”

setMode

Class: cgv.CGV
Package: cgv

Specify mode of execution

Syntax

```
cgvObj.setMode(connectivity)
```

Description

`cgvObj.setMode(connectivity)` specifies the mode of execution for the `cgv.CGV` object, *`cgvObj`*. The default value for the execution mode is set to either `normal` or `sim`.

Input Arguments

`connectivity`

Specify mode of execution

Value	Description
`sim` or `normal` (default)	Mode of execution is normal simulation.
`sil`	Mode of execution is SIL.
`pil`	Mode of execution is PIL.

Examples

After running a `cgv.CGV` object, copy the object. Before rerunning the object, call `setMode` to change the execution mode to `sil` for an existing `cgv.CGV` object.

```
cgvModel = 'rtwdemo_cgv';
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');
```

```
cgvObj1.run();
cgvObj2 = cgvObj1.copySetup()
cgvObj2.setMode('sil');
cgvObj2.run();
```

See Also

`cgv.CGV.copySetup` | `cgv.CGV.run`

Topics

“Verify Numerical Equivalence with CGV”

setNameSpace

Set namespace for C++ function entry in code replacement table

Syntax

```
setNameSpace(hEntry,nameSpace)
```

Description

`setNameSpace(hEntry,nameSpace)` specifies the namespace for a C++ function entry in a code replacement table.

During code generation, if the function entry is matched, the software emits the namespace in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`).

If you created the function entry by using *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry = MyCustomFunctionEntry`* (did not use `registerCPPFunctionEntry`), before calling the `setNameSpace` function, enable C++ support for the function entry by calling the `enableCPP` function.

Examples

Set Namespace for Implementation Function

This example shows how to use the `setNameSpace` function to set the namespace for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                 'Key',                    'sin', ...
                                 'Priority',               100, ...
                                 'ImplementationName',     'sin', ...
                                 'ImplementationHeaderFile', 'cmath' );
```

```
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

Input Arguments

`hEntry` — Handle to a code replacement function entry
handle

The *`hEntry`* is a handle to a code replacement function entry previously returned by one of the following:

- *`hEntry`* `= RTW.TflCFunctionEntry`
- *`hEntry`* `= MyCustomFunctionEntry`, where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`
- A call to the `registerCPPFunctionEntry` function

Example: `fcn_entry`

`nameSpace` — Specifies the namespace in which the implementation function for the C++ function entry is defined
character vector | string scalar

Example: `'std'`

See Also
`enableCPP` | `registerCPPFunctionEntry`

Topics
“Define Code Replacement Mappings”

Introduced in R2010a

setNamespace

Class: RTW.ModelCPPClass
Package: RTW

Set namespace in model-specific C++ class interface

Syntax

`setNamespace(obj, nsName)`

Description

`setNamespace(obj, nsName)` sets the namespace in the specified model-specific C++ class interface.

Input Arguments

obj	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *obj* = RTW.ModelCPPArgsClass on page 1-412, *obj* = RTW.ModelCPPDefaultClass on page 1-418, or *obj* = RTW.getClassInterfaceSpecification (*modelName*).
nsName	Character vector specifying a namespace for the class described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the model namespace in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the namespace for your model class. In the I/O arguments step

`method` view of this dialog box, click the **Get Default Configuration** button to display the model namespace, which you can examine and modify. In the `Default step method` view, you can examine and modify the model namespace without having to click a button. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

setObjectiveName

Class: rtw.codegenObjectives.Objective
Package: rtw.codegenObjectives

Specify objective name

Syntax

`setObjectiveName(`*`obj`*`, `*`objName`*`)`

Description

`setObjectiveName(`*`obj`*`, `*`objName`*`)` specifies a name for the objective. The Configuration Set Objectives dialog box displays the name of the objective.

Input Arguments

`obj`	Handle to a code generation objective object previously created.
`objName`	Optional character vector that indicates the name of the objective. If you do not specify an objective name, the Configuration Set Objectives dialog box displays the objective ID for the objective name.

Examples

Name the objective Reduce RAM Example:

```
setObjectiveName(obj, 'Reduce RAM Example');
```

See Also

Topics

“Create Custom Code Generation Objectives”

setOutputDir

Class: cgv.CGV
Package: cgv

Specify folder

Syntax

```
cgvObj.setOutputDir('path')
cgvObj.setOutputDir('path', 'overwrite', 'on')
```

Description

`cgvObj.setOutputDir('path')` is an optional method that specifies a location where the object writes the output and metadata files for execution. *`cgvObj`* is a handle to a `cgv.CGV` object. *`path`* is the absolute or relative path to the folder. If the path does not exist, the object attempts to create the folder. If you do not call `setOutputDir`, the object uses the current working folder.

`cgvObj.setOutputDir('path', 'overwrite', 'on')` includes the property and value pair to allow read-only files in the working directory to be overwritten. The default value for `'overwrite'` is `'off'`.

See Also

Topics

“Verify Numerical Equivalence with CGV”

setOutputFile

Class: cgv.CGV
Package: cgv

Specify output data file name

Syntax

`cgvObj.setOutputFile(InputIndex,OutputFile)`

Description

`cgvObj.setOutputFile(InputIndex,OutputFile)` is an optional method that changes the default file name for the output data. *`cgvObj`* is a handle to a `cgv.CGV` object. *`InputIndex`* is a unique numeric identifier that specifies which output data to write to the file. The *`InputIndex`* is associated with specific input data.*`OutputFile`* is the name of the file, with or without the `.mat` extension.

See Also

Topics

“Verify Numerical Equivalence with CGV”

setReservedIdentifiers

Register reserved identifiers to associate with code replacement library

Syntax

```
setReservedIdentifiers(hTable,ids)
```

Description

`setReservedIdentifiers(hTable,ids)` registers reserved identifier structures in a code replacement table.

In a code replacement table, the code generator registers each function implementation name defined by a table entry as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

The `setReservedIdentifiers` function lets you register up to four reserved identifier structures in a code replacement table. One set of reserved identifiers can be associated with a code replacement library, while the other three (if present) must be associated with libraries named `ANSI_C`, `ISO_C`, `ISO_C++`, or `GNU`.

For information about generating a list of reserved identifiers for the code replacement library that you use to generate code, see “Reserved Identifiers and Code Replacement”.

Examples

Register Reserved Identifier Structures

This example shows how to use the `setReservedIdentifiers` function to register four reserved identifier structures, for `'ANSI_C'`, `'ISO_C'`,`'ISO_C++'`, and `'My Custom CRL'`, respectively.

```
hLib = RTW.TflTable;

% Create and register CRL entries here

.
.
.

% Create and register reserved identifiers
d{1}.LibraryName = 'ANSI_C';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{1}.HeaderInfos{2}.HeaderName = 'foo.h';
d{1}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{2}.LibraryName = 'ISO_C';
d{2}.HeaderInfos{1}.HeaderName = 'math.h';
d{2}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{2}.HeaderInfos{2}.HeaderName = 'foo.h';
d{2}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{3}.LibraryName = 'ISO_C++';
d{3}.HeaderInfos{1}.HeaderName = 'math.h';
d{3}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{3}.HeaderInfos{2}.HeaderName = 'foo.h';
d{3}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{4}.LibraryName = 'My Custom CRL';
d{4}.HeaderInfos{1}.HeaderName = 'my_math_lib.h';
d{4}.HeaderInfos{1}.ReservedIds = {'y1', 'u1'};
d{4}.HeaderInfos{2}.HeaderName = 'my_oper_lib.h';
d{4}.HeaderInfos{2}.ReservedIds = {'foo', 'bar'};

setReservedIdentifiers(hLib, d);
```

Input Arguments

hTable — Handle to a code replacement table

handle

The *hTable* is a handle to a code replacement table previously returned by *hTable* = RTW.TflTable.

Example: `hLib`

`ids` — Specifies reserved keywords to register for library
structure

The *`ids`* is a structure specifying reserved keywords to be registered for a library. The structure must contain:

- `LibraryName` element, a character vector or string scalar that specifies `'ANSI_C'`, `'ISO_C'`, `'ISO_C++'`, `'GNU'`.
- `HeaderInfos` element, a structure or cell array of structures containing:
  - `HeaderName` element, a character vector or string scalar that specifies the header file in which the identifiers are declared.
  - `ReservedIds` element, a cell array of character vectors or string array that specifies the names of the identifiers to be registered as reserved keywords.

Example: `d`

See Also

Topics

“Reserved Identifiers and Code Replacement”

Introduced in R2008a

setStepMethodName

Class: RTW.ModelCPPClass
Package: RTW

Set step method name in model-specific C++ class interface

Syntax

`setStepMethodName(`*`obj`*`, `*`fcnName`*`)`

Description

`setStepMethodName(`*`obj`*`, `*`fcnName`*`)` sets the step method name in the specified model-specific C++ class interface.

Input Arguments

`obj`	Handle to a model-specific C++ class interface control object, such as a handle previously returned by *`obj`* `= RTW.ModelCPPArgsClass on page 1-412`, *`obj`* `= RTW.ModelCPPDefaultClass on page 1-418`, or *`obj`* `= RTW.getClassInterfaceSpecification (`*`modelName`*`)`.
`fcnName`	Character vector specifying a new name for the step method described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments`

`step method` view of this dialog box, click the **Get Default Configuration** button to display the step method name, which you can examine and modify. In the `Default step method` view, you can examine and modify the step method name without having to click a button. For more information, see “Configure Step Method for Your Model Class”.

See Also

Topics

“Customize C++ Class Interfaces Programmatically”
“Configure Step Method for Model Class”
“Customize Generated C++ Class Interfaces”

setTflCFunctionEntryParameters

Set specified parameters for function entry in code replacement table

Syntax

```
setTflCFunctionEntryParameters(hEntry,varargin)
```

Description

`setTflCFunctionEntryParameters(hEntry,varargin)` sets specified parameters for a function entry in a code replacement table.

Examples

Specify Parameters for Function Entry

This example shows how to use the `setTflCFunctionEntryParameters` function to set specified parameters for a code replacement function entry for `sqrt`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                'Key',                    'sqrt', ...
                                'Priority',               100, ...
                                'ImplementationName',     'sqrt', ...
                                'ImplementationHeaderFile', '<math.h>' );
```

Input Arguments

`hEntry` — Handle to a code replacement function entry

handle

The *hEntry* is a handle to a code replacement function entry previously returned by *hEntry* = RTW.TflCFunctionEntry or *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from RTW.TflCFunctionEntry.

Example: fcn_entry

varargin — Name-value pairs of arguments for function entry
name-value pairs

Example: 'Key','sqrt'

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Key','sqrt'

AcceptExprInput — Selects whether implementation function accepts expression inputs
true | false

The *AcceptExprInput* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is true if ImplType equals FCN_IMPL_FUNCT and false if ImplType equals FCN_IMPL_MACRO.

If the value is true, expression inputs are integrated into the generated code in a form similar to this form:

```
 rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is false, a temporary variable is generated for the expression input:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

Example: 'AcceptExprInput',true

AdditionalHeaderFiles — Specifies additional header files for table entry
{} (default) | array of character vectors | string array

The *`AdditionalHeaderFiles`* value specifies additional header files for a code replacement table entry. The vectors can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

`AdditionalIncludePaths` — Specifies additional include paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalIncludePaths`* value specifies the full path of additional include paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

`AdditionalLinkObjs` — Specifies additional link objects for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjs`* value specifies additional link objects for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

`AdditionalLinkObjsPaths` — Specifying additional link object paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjsPaths`* value specifies the full path of additional link object paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector. The default is `{}`.

Example: `'AdditionalLinkObjsPaths',{}`

`AdditionalSourceFiles` — Specifies additional source files for table entry
`{}` (default) | array of character vectors | string array

The *AdditionalSourceFiles* value specifies additional source files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

AdditionalSourcePaths — Specifies additional source paths for table entry
`{}` (default) | array of character vectors | string array

The *AdditionalSourcePaths* value specifies the full path of additional source paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourcePaths',{}`

AdditionalCompileFlags — Specifies additional compiler flags for table entry
`{}` (default) | array of character vectors | string array

The *AdditionalCompileFlags* value specifies additional flags required to compile the source files defined for a code replacement table entry. The default is `{}`.

Example: `'AdditionalCompileFlags',{}`

AdditionalLinkFlags — Specifies additional linker flags for table entry
`{}` (default) | array of character vectors | string array

The *AdditionalLinkFlags* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

ArrayLayout — Specifies layout of array storage for table entry
`'COLUMN_MAJOR'` (default) | `'ROW_MAJOR'` | `'COLUMN_AND_ROW'`

The *ArrayLayout* value specifies the order of array elements in memory supported by the replacement implementation. By default, the replacement implementation supports column-major data layout. For `ROW-MAJOR`, the replacement implementation supports row-major data layout. For `COLUMN_AND_ROW`, the replacement implementation supports column-major and row-major data layouts.

Example: `'ArrayLayout','ROW_MAJOR'`

`EntryInfoAlgorithm` — Specifies computation or approximation method to match for table entry
`'RTW_DEFAULT'` | `'RTW_NEWTON_RAPHSON'` | `'RTW_CORDIC'` | `'RTW_UNSPECIFIED'`

The *`EntryInfoAlgorithm`* value specifies a computation or approximation method, configured for the specified math function, that must be matched in order for function replacement to occur. Code replacement libraries support function replacement based on computation or approximation method for the math functions `rSqrt`, `sin`, `cos`, and `sincos`. The valid arguments for each supported function are listed in the table.

Function	Argument	Meaning
`rSqrt`	`RTW_DEFAULT`	Match the default computation method, `Exact`
	`RTW_NEWTON_RAPHSON`	Match the `Newton-Raphson` computation method
	`RTW_UNSPECIFIED`	Match a computation method
`sin` `cos` `sincos`	`RTW_CORDIC`	Match the `CORDIC` approximation method
	`RTW_DEFAULT`	Match the default approximation method, `None`
	`RTW_UNSPECIFIED`	Match an approximation method

Example: `'EntryInfoAlgorithm','RTW_DEFAULT'`

`GenCallback` — Specifies callback that follows code generation
`''` (default) | `'RTW.copyFileToBuildDir'`

The *`GenCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `'GenCallback',''`

`ImplementationHeaderFile` — Specifies the name of the header file that declares the implementation function
`''` (default) | character vector | string scalar

The *ImplementationHeaderFile* value specifies the name of the header file that declares the implementation function, for example, `'<math.h>'`. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile',''`

ImplementationHeaderPath — Specifies path to implementation header file
`''` (default) | character vector | string scalar

The *ImplementationHeaderPath* value specifies the full path to the implementation header file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath',''`

ImplementationName — Specifies name of implementation function
`''` (default) | character vector | string scalar

The *ImplementationName* value specifies the name of the implementation function, for example, `'sqrt'`, which can match or differ from the `Key` name.

Example: `'ImplementationName',''`

ImplementationSourceFile — Specifies name of implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourceFile* value specifies the name of the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile',''`

ImplementationSourcePath — Specifies path to implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourcePath* value specifies the full path to the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in

the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath',''`

`ImplType` — Specifies the type of entry
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

`Key` — Specifies name of function to replace
character vector | string scalar

The Key value specifies the name of the function to replace. The name must match a function name listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'Key','sqrt'`

`Priority` — Specifies the search priority for function entry
`100` (default) | integer 0..100

The *`Priority`* value specifies the search priority for the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

`RoundingModes` — Specifying rounding modes supported by implementation function
`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_SIMPLEST'` | `'RTW_ROUND_CONV'` | array of character vectors | string array

The *`RoundingModes`* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

`SaturationMode` — Specifying saturation mode supported by implementation function

`'RTW_SATURATE_UNSPECIFIED'` (default) | `'RTW_SATURATE_ON_OVERFLOW'` | `'RTW_WRAP_ON_OVERFLOW'`

The *`SaturationMode`* value specifies the saturation mode supported by the implementation function.

Example: `'SaturationMode','RTW_SATURATE_UNSPECIFIED'`

`SideEffects` — Specifies whether to attempt to optimize away the implementation function

`false` (default) | `true`

The *`SideEffects`* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return `void` but are not to be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`.

Example: `'SideEffects',false`

`StoreFcnReturnInLocalVar` — Specifies whether to store the implementation function regardless expression folding settings

`false` (default) | `true`

The *`StoreFcnReturnInLocalVar`* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false`, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

setTflCOperationEntryParameters

Set specified parameters for operator entry in code replacement table

Syntax

```
setTflCOperationEntryParameters(hEntry,varargin)
```

Description

`setTflCOperationEntryParameters(hEntry,varargin)` sets specified parameters for an operator entry in a code replacement table.

Examples

Set Parameters for Addition Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for `uint8` addition that matches a cast-after-sum algorithm.

```
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'EntryInfoAlgorithm',       'RTW_CAST_AFTER_OP', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_SATURATE_UNSPECIFIED', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName',       'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c');
```

Set Parameters for Fixed-Point Division Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for fixed-point `int16` division. The table entry specifies a net scaling between the operator inputs and output to map a range of slope and bias values to a replacement operation.

```
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_DIV', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_CEILING'}, ...
    'NetSlopeAdjustmentFactor',   1.0, ...
    'NetFixedExponent',           0.0, ...
    'ImplementationName',       's16_div_s16_s16', ...
    'ImplementationHeaderFile', 's16_div_s16_s16.h', ...
    'ImplementationSourceFile', 's16_div_s16_s16.c' );
```

Set Parameters for Fixed-Point Addition Operator Entry

This example shows how to use the `setTflCOperationEntryParameters` function to set parameters for a code replacement operator entry for fixed-point `uint16` addition that matches a cast-after-sum algorithm. The parameters 'SlopesMustBeTheSame' and 'MustHaveZeroNetBias' must be set to `true` to specify equal slope and zero net bias across operator inputs and output. This maps relative slope and bias values (rather than a specific slope and bias combination) to a replacement operation.

```
op_entry = RTW.TflCOperationEntryGenerator;
op_entry.setTflCOperationEntryParameters( ...
    'Key',                      'RTW_OP_ADD', ...
    'EntryInfoAlgorithm',       'RTW_CAST_AFTER_OP', ...
    'Priority',                 90, ...
    'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
    'RoundingModes',            {'RTW_ROUND_UNSPECIFIED'}, ...
    'SlopesMustBeTheSame',      true, ...
    'MustHaveZeroNetBias',      true, ...
    'ImplementationName',       'u16_add_SameSlopeZeroBias', ...
```

```
'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
```

Input Arguments

hEntry — Handle to a code replacement table entry

handle

The *hEntry* is a handle to a code replacement table entry previously returned by one of the class instantiations in the table.

Class Instantiation	Support
hEntry = RTW.TflCOperationEntry;	Supports operator replacement.
hEntry = RTW.TflCOperationEntryGenerator;	Provides parameters for fixed-point addition and subtraction that are not available in RTW.TflCOperationEntry (SlopesMustBeTheSame and MustHaveZeroNetBias).
hEntry = RTW.TflCOperationEntryGenerator_NetSlope;	Provides net slope parameters for fixed-point multiplication and division that are not available in RTW.TflCOperationEntry (NetSlopeAdjustmentFactor and NetFixedExponent).
hEntry = RTW.TflBlasEntryGenerator;	Supports replacement of nonscalar operators with MathWorks BLAS functions.
hEntry = RTW.TflCBlasEntryGenerator;	Supports replacement of nonscalar operators with ANSI/ISO® C BLAS functions.
hEntry = *MyCustomOperationEntry*; (where *MyCustomOperationEntry* is a class derived from RTW.TflCOperationEntry)	Supports operator replacement using custom code replacement table entries.

If you want to specify SlopesMustBeTheSame or MustHaveZeroNetBias for your operator entry, instantiate your table entry using *hEntry* =

`RTW.TflCOperationEntryGenerator` rather than *`hEntry`* = `RTW.TflCOperationEntry`. If you want to use `NetSlopeAdjustmentFactor` and `NetFixedExponent`, instantiate your table entry by using *`hEntry`* = `RTW.TflCOperationEntryGenerator_NetSlope`.

Example: `op_entry`

`varargin` — Name-value pairs of arguments for operation entry
name-value pairs

Example: `'Key','RTW_OP_ADD'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Key','RTW_OP_ADD'`

`AcceptExprInput` — Specifies whether implementation operation accepts expression inputs
`true` | `false`

The *`AcceptExprInput`* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

If the value is `true`, expression inputs are integrated into the generated code in a form similar to this form:

```
 rtY.Out1 = myAdd(rtU.In1, rtU.In2 * rtU.In3);
```

If the value is `false`, a temporary variable is generated for the expression input:

```
real_T temp;

temp = rtU.In2 * rtU.In3;
rtY.Out1 = myAdd(rtU.In1, temp);
```

Example: `'AcceptExprInput',true`

`AdditionalHeaderFiles` — Specifies additional header files for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalHeaderFiles`* value specifies additional header files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

`AdditionalIncludePaths` — Specifies additional include paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalIncludePaths`* value specifies the full path of additional include paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

`AdditionalLinkObjs` — Specifies additional link objects for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjs`* value specifies additional link objects for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

`AdditionalLinkObjsPaths` — Specifies additional link object paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjsPaths`* value specifies the full path of additional link object paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjsPaths',{}`

`AdditionalSourceFiles` — specifies additional source files for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourceFiles`* value specifies additional source files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

`AdditionalSourcePaths` — Specifies additional source paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourcePaths`* value specifies the full path of additional source paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourcePaths',{}`

`AdditionalCompileFlags` — Specifies additional compiler flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalCompileFlags`* value specifies additional flags required to compile the source files defined for a code replacement table entry.

Example: `'AdditionalCompileFlags',{}`

`AdditionalLinkFlags` — Specifies additional linker flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkFlags`* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

`AllowShapeAgnosticMatch` — Enables matrix matches based on the total number of elements rather than specific matrix shape
`false` (default) | `true`

The *AllowShapeAgnosticMatch* value enables code replacement match based on total number of elements rather than specific matrix shape for matrices that are contiguously allocated in memory. For more information, see “Allow Shape Agnostic Match”.

Example: `'AllowShapeAgnosticMatch', false`

`ArrayLayout` — Specifies layout of array storage for table entry
`'COLUMN_MAJOR'` (default) | `'ROW_MAJOR'` | `'COLUMN_AND_ROW'`

The *ArrayLayout* value specifies the order of array elements in memory supported by the replacement implementation. By default, the replacement implementation supports column-major data layout. For `ROW-MAJOR`, the replacement implementation supports row-major data layout. For `COLUMN_AND_ROW`, the replacement implementation supports column-major and row-major data layouts.

Example: `'ArrayLayout','ROW_MAJOR'`

`EntryInfoAlgorithm` — Specifies math algorithm to match for table entry
`'RTW_CAST_BEFORE_OP'` (default) | `'RTW_CAST_AFTER_OP'`

The *EntryInfoAlgorithm* value specifies the algorithm for the specified math function that must be matched for operator replacement to occur. Code replacement libraries support replacement based on the algorithm for math operations `RTW_OP_ADD` and `RTW_OP_MINUS`. Valid arguments for the supported operations are listed in the table. The arguments have the same meaning for both operations.

Argument	Meaning
`RTW_CAST_BEFORE_OP`	Before performing the operation, type cast input values to the output data type. If the output type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type.

Argument	Meaning
RTW_CAST_AFTER_OP	Compute the ideal result of the operation of inputs. Then, type cast the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operation to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal.

Example: `'EntryInfoAlgorithm','RTW_CAST_AFTER_OP'`

`GenCallback` — Specifies callback that follows code generation
`''` (default) | `'RTW.copyFileToBuildDir'`

The *`GenCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this operation entry is matched and used, the code generator calls function `RTW.copyFileToBuildDir` after code generation. This callback function copies additional header, source, or object files that you have specified for this entry to the build folder. For more information, see “Specify Build Information for Replacement Code”.

Example: `'GenCallback','RTW.copyFileToBuildDir'`

`ImplementationHeaderFile` — Specifies name of header file that declares implementation operation
`''` (default) | character vector | string scalar

The *`ImplementationHeaderFile`* value specifies the name of the header file that declares the implementation function. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile','s32_mul.h'`

`ImplementationHeaderPath` — Specifies path to implementation header file
`''` (default) | character vector | string scalar

The *ImplementationHeaderPath* value specifies the full path to the implementation header file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath', fullfile('$MATLAB_ROOT', 'crl')`

`ImplementationName` — Specifies name of implementation function
`''` (default) | character vector | string scalar

The *ImplementationName* value specifies the name of the implementation function, which can match or differ from the `Key` name.

Example: `'ImplementationName','s32_mul_s32_s32_sat'`

`ImplementationSourceFile` — specifies name of implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourceFile* value specifies the name of the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile','s32_mul.c'`

`ImplementationSourcePath` — Specifies path to implementation source file
`''` (default) | character vector | string scalar

The *ImplementationSourcePath* value specifies the full path to the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath', fullfile('$MATLAB_ROOT', 'crl')`

`ImplType` — Specifies whether the table entry is for an implementation function or macro
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

The *ImplType* value specifies the type of table entry. Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

Key — Specifies key for operator to replace
character vector | string scalar

The *`Key`* value specifies the key for the operator to replace. The key must match an operator key listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'Key','RTW_OP_ADD'`

`MustHaveZeroNetBias` — Specifies net bias requirement for conceptual arguments of Add/Minus entries
`false` (default) | `true`

The *`MustHaveZeroNetBias`* value specifies whether a replacement match requires that the net bias for conceptual arguments of Add/Minus entries is zero. For the Add/Minus of fixed-point operator inputs and output this parameter must be set to `true`. Instantiate the entry by using *`hEntry`* `= RTW.TflCOperationEntryGenerator` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

For Mul/Div/MulDiv/Shift/Cast entries:

- The code generator ignores the value of this argument.
- The bias of the conceptual arguments for Mul/Div/MulDiv/Shift/Cast entries must be zero for a replacement match to occur.

Example: `'MustHaveZeroNetBias',true`

`NetFixedExponent` — Specifies fixed exponent part of net slope for fixed-point conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries
`0` (default) | `numeric scalar`

The *`NetSlopeAdjustmentFactor`* value specifies the fixed exponent (E) part of the net slope ($F2^E$, for example, -3.0) for fixed-point conceptual arguments required for a replacement match to occur for Mul/Div/MulDiv/Shift/Cast entries. Instantiate an entry by using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

For Add/Minus entries:

- The code generator ignores the value of this argument.

- The slope adjustment factor part of the net slope of the conceptual arguments for Add/Minus entries must be zero for a replacement match to occur.

Example: `'NetFixedExponent', -3.0`

`NetSlopeAdjustmentFactor` — Specifies slope adjustment part of net slope requirement for fixed-point conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries

`1` (default) | numeric scalar

The *`NetSlopeAdjustmentFactor`* value specifies the slope adjustment part of the net slope ($F2^E$, for example, 1.0) for fixed-point conceptual arguments required for a replacement match to occur for Mul/Div/MulDiv/Shift/Cast entries. Instantiate an entry by using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

For Add/Minus entries:

- The code generator ignores the value of this argument.
- The slope adjustment factor part of the net slope of the conceptual arguments for Add/Minus entries must be zero for a replacement match to occur.

Example: `'NetSlopeAdjustmentFactor',1.5`

`Priority` — Specifies the search priority of operator entry

`100` (default) | integer value 0..100

The *`Priority`* value specifies the search priority of the operation entry, relative to other entries of the same operation name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for an operation, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

`RoundingModes` — Specifies rounding modes supported by implementation function

`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_CONV'` | `'RTW_ROUND_SIMPLEST'` | array of character vectors | string array

The *RoundingModes* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

SaturationMode — specifies saturation mode supported by implementation function
`'RTW_SATURATE_UNSPECIFIED'` (default) | `'RTW_SATURATE_ON_OVERFLOW'` | `'RTW_WRAP_ON_OVERFLOW'` | character vector | string scalar

The *SaturationMode* value specifies the saturation mode supported by the implementation function.

Example: `'SaturationMode','RTW_SATURATE_UNSPECIFIED'`

SideEffects — Specifies whether to attempt to optimize away the implementation function
`false` (default) | `true`

The *SideEffects* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return `void` but are not to be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`.

Example: `'SideEffects',false`

SlopesMustBeTheSame — Specifies slope requirement for conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries
`false` (default) | `true`

The *SlopesMustBeTheSame* value specifies whether a replacement match requires that the slope is the same for conceptual arguments of Mul/Div/MulDiv/Shift/Cast entries. Instantiate the entry by using *hEntry* `= RTW.TflCOperationEntryGenerator` rather than *hEntry* `= RTW.TflCOperationEntry`.

For Add/Minus entries:

- The code generator ignores the value of this argument.
- This parameter must be set to `true`. When set to `true`, the slopes of the conceptual arguments are equal for a replacement match to occur.

Example: `'SlopesMustBeTheSame',true`

`StoreFcnReturnInLocalVar` — Specifies whether to store the implementation function regardless expression folding settings
`false` (default) | `true`

The *`StoreFcnReturnInLocalVar`* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false`, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”
“Scalar Operator Code Replacement”
“Addition and Subtraction Operator Code Replacement”
“Small Matrix Operation to Processor Code Replacement”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2007b

setTflCSemaphoreEntryParameters

Set specified parameters for semaphore entry in code replacement table

Syntax

```
setTflCSemaphoreEntryParameters(hEntry,varargin)
```

Description

`setTflCSemaphoreEntryParameters(hEntry,varargin)` sets specified parameters for a semaphore entry in a code replacement table.

Examples

Specify Semaphore Initialization Parameters for Table Entry

This example shows how to use the `setTflCSemaphoreEntryParameters` function to set specified parameters for a code replacement table entry for a semaphore initialization replacement.

```
sem_entry = RTW.TflCSemaphoreEntry;
sem_entry.setTflCSemaphoreEntryParameters( ...
    'Key',                      'RTW_SEM_INIT', ...
    'Priority',                 100, ...
    'ImplementationName',       'mySemCreate', ...
    'ImplementationHeaderFile', 'mySem.h', ...
    'ImplementationSourceFile', 'mySem.c', ...
```

```
    'GenCallback',              'RTW.copyFileToBuildDir', ...
    'SideEffects',              true);
```

Input Arguments

`hEntry` — Handle to semaphore entry

handle

The *`hEntry`* is a handle to a code replacement library semaphore entry previously returned by *`hEntry`* `= RTW.TflCSemaphoreEntry;`.

Example: `sem_entry`

`varargin` — Name-value pairs of arguments for function entry

name-value pairs

Example: `'Key','RTW_SEM_INIT'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Key','RTW_SEM_INIT'`

`AcceptExprInput` — Specifies whether implementation function accepts expression inputs

`true` | `false`

The *`AcceptExprInput`* value flags the code generator that the implementation function described by this entry accepts expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

If the value is `true`, expression inputs are integrated into the generated code in a form similar to this form:

```
rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is `false`, a temporary variable is generated for the expression input:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

Example: `'AcceptExprInput',true`

`AdditionalHeaderFiles` — Specifies additional header files for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalHeaderFiles`* value specifies additional header files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalHeaderFiles',{}`

`AdditionalIncludePaths` — Specifies additional include paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalIncludePaths`* value specifies the full path of additional include paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalIncludePaths',{}`

`AdditionalLinkObjs` — Specifies additional link objects for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjs`* value specifies additional link objects for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjs',{}`

`AdditionalLinkObjsPaths` — Specifies additional link object paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkObjsPaths`* value specifies the full path of additional link object paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalLinkObjsPaths',{}`

`AdditionalSourceFiles` — specifies additional source files for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourceFiles`* value specifies additional source files for a code replacement table entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'AdditionalSourceFiles',{}`

`AdditionalSourcePaths` — Specifies additional source paths for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalSourcePaths`* value specifies the full path of additional source paths for a code replacement entry. The character vectors or string array can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector. The default is `{}`.

Example: `'AdditionalSourcePaths',{}`

`AdditionalCompileFlags` — Specifies additional compiler flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalCompileFlags`* value specifies additional flags required to compile the source files defined for a code replacement table entry.

Example: `'AdditionalCompileFlags',{}`

`AdditionalLinkFlags` — Specifies additional linker flags for table entry
`{}` (default) | array of character vectors | string array

The *`AdditionalLinkFlags`* value specifies additional flags required to link the compiled files for a code replacement table entry.

Example: `'AdditionalLinkFlags',{}`

`GenCallback` — Specifies callback that follows code generation
`''` (default) | `'RTW.copyFileToBuildDir'`

The *`GenCallback`* specifies a callback that follows code generation. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` is called after code generation. This callback function copies additional header, source, or object files that you have specified for this function entry to the build folder. For more information, see "Specify Build Information for Replacement Code".

Example: `'GenCallback',''`

`ImplementationHeaderFile` — Specifies name of header file that declares implementation function
`''` (default) | character vector | string scalar

The *`ImplementationHeaderFile`* value specifies the name of the header file that declares the implementation function. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderFile','<math.h>'`

`ImplementationHeaderPath` — Specifies path to implementation header file
`''` (default) | character vector | string scalar

The *`ImplementationHeaderPath`* value specifies the full path to the implementation header file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationHeaderPath',''`

`ImplementationName` — Specifies name of implementation function
`''` (default) | character vector | string scalar

The *`ImplementationName`* value specifies the name of the implementation function, which can match or differ from the `Key` name.

Example: `'ImplementationName','sqrt'`

`ImplementationSourceFile` — specifies name of implementation source file
`''` (default) | character vector | string scalar

The *`ImplementationSourceFile`* value specifies the name of the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourceFile',''`

`ImplementationSourcePath` — Specifies path to implementation source file
`''` (default) | character vector | string scalar

The *`ImplementationSourcePath`* value specifies the full path to the implementation source file. The character vector or string scalar can include tokens. For example, in the token `$mytoken$`, `mytoken` is a variable defined as a character vector or string scalar in the MATLAB workspace or as a MATLAB function in the search path that returns a character vector.

Example: `'ImplementationSourcePath',''`

`ImplType` — Specifies the type of table entry
`'FCN_IMPL_FUNCT'` (default) | `'FCN_IMPL_MACRO'`

The *`ImplType`* value specifies the type of table entry. Use `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Example: `'ImplType','FCN_IMPL_FUNCT'`

`Key` — Specifies key for operator to replace
character vector | string scalar

The *`Key`* value specifies the key for the operator to replace. The name must match a function name listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

Example: `'Key','RTW_OP_ADD'`

`Priority` — Specifies the search priority of the function entry
`100` (default) | integer value 0..100

The *`Priority`* value specifies the search priority of the function entry, relative to other entries of the same function name and conceptual argument list within this table. Highest

priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for a function, the implementation with the higher priority shadows the one with the lower priority.

Example: `'Priority',100`

`RoundingModes` — Specifies rounding modes supported by implementation function

`'RTW_ROUND_UNSPECIFIED'` (default) | `'RTW_ROUND_FLOOR'` | `'RTW_ROUND_CEILING'` | `'RTW_ROUND_ZERO'` | `'RTW_ROUND_NEAREST'` | `'RTW_ROUND_NEAREST_ML'` | `'RTW_ROUND_CONV'` | `'RTW_ROUND_SIMPLEST'` | array of character vectors | string array

The *`RoundingModes`* value specifies one or more rounding modes supported by the implementation function.

Example: `'RoundingModes',{'RTW_ROUND_UNSPECIFIED'}`

`SaturationMode` — specifies saturation mode supported by implementation function

`'RTW_SATURATE_UNSPECIFIED'` (default) | `'RTW_SATURATE_ON_OVERFLOW'` | `'RTW_WRAP_ON_OVERFLOW'` | character vector | string scalar

The *`SaturationMode`* value specifies the saturation mode supported by the implementation function.

Example: `'SaturationMode','RTW_SATURATE_UNSPECIFIED'`

`SideEffects` — Specifies whether to attempt to optimize away the implementation function

`false` (default) | `true`

The *`SideEffects`* value flags the code generator not to optimize away the implementation function described by this entry. This parameter applies to implementation functions that return `void` but are not to be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`.

Example: `'SideEffects',false`

`StoreFcnReturnInLocalVar` — Specifies whether to store the implementation function regardless expression folding settings

`false` (default) | `true`

The *`StoreFcnReturnInLocalVar`* value flags the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false`, other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. This example shows code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5),sw_U.In6)) {
      sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code is potentially easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    ......
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
       sw_Y.Out1 = sw_U.In7;
    } else {
       sw_Y.Out1 = sw_U.In8;
    }
}
```

Example: `'StoreFcnReturnInLocalVar',false`

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalLinkObj` | `addAdditionalLinkObjPath` | `addAdditionalSourceFile` | `addAdditionalSourcepath`

Topics

“Specify Build Information for Replacement Code”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”
“Mutex and Semaphore Functions”

Introduced in R2013a

coder.MATLABCodeTemplate.setTokenValue

Class: coder.MATLABCodeTemplate
Package: coder

Set value of token for code generation template

Syntax

```
setTokenValue(tokenName,tokenValue)
```

Description

`setTokenValue(tokenName,tokenValue)` sets the value of a token for a code generation template.

Input Arguments

`tokenName`

The name of the token

Default:

`tokenValue`

The value of the token

Default: empty

Examples

Create a MATLABCodeTemplate object from a custom template. Set the value for a custom token in the template.

```
newObj = coder.MATLABCodeTemplate('myCGTFile');
% Create a MATLABCodeTemplate object from a custom template file
newObj.setTokenValue('myCustomToken', 'myValue');
% Set the value of a custom token in the file
newObj.getTokenValue('myCustomToken')
% Check value of the custom token
```

See Also

`coder.MATLABCodeTemplate.emitSection` |
`coder.MATLABCodeTemplate.getCurrentTokens` |
`coder.MATLABCodeTemplate.getTokenValue`

Topics

“Generate Custom File and Function Banners for C/C++ Code”
“Code Generation Template Files for MATLAB Code”

setAlgorithmParameters

Set algorithm parameters for lookup table function code replacement table entry

Syntax

```
setAlgorithmParameters(tableEntry, algParams)
```

Description

`setAlgorithmParameters(tableEntry, algParams)` sets the algorithm parameters for the lookup table function identified in the code replacement table entry `tableEntry`.

Examples

Set Algorithm Parameters for `prelookup` Function Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `prelookup` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'prelookup', ...
    'Priority', 100, ...
    'ImplementationName', 'Ifx_DpSearch_u8');
```

Get the algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams =

  Prelookup with properties:

            ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
```

```
                RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
      IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
      UseLastBreakpoint: [1x1 coder.algorithm.parameter.UseLastBreakpoint]
  RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
```

Display the valid values for parameter `UseLastBreakpoint` for the `prelookup` function.

```
algParams.UseLastBreakpoint
```

```
ans =

  UseLastBreakpoint with properties:

       Name: 'UseLastBreakpoint'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Display the valid values for parameter `RemoveProtectionInput` for the `prelookup` function.

```
algParams.RemoveProtectionInput
```

```
ans =

  RemoveProtectionInput with properties:

       Name: 'RemoveProtectionInput'
    Options: {'off'  'on'}
    Primary: 0
      Value: {'off'  'on'}
```

Set parameters `UseLastBreakpoint` and `RemoveProtectionInput` to `on` and `off`, respectively.

```
algParams.UseLastBreakpoint = 'on';
algParams.RemoveProtectionInput = 'off';
```

When you set each parameter, the algorithm parameter software checks for and reports errors for invalid syntax, parameter names, and values.

Update the parameter settings for the code replacement table entry.

```
setAlgorithmParameters(tableEntry, algParams);
```

Get the new algorithm parameter settings for the `prelookup` function table entry.

```
algParams = getAlgorithmParameters(tableEntry);
```

Examine the new value for `UseLastBreakpoint`.

```
algParams.UseLastBreakpoint
```

```
ans =

  UseLastBreakpoint with properties:
```

```
      Name: 'UseLastBreakpoint'
   Options: {'off'  'on'}
   Primary: 0
     Value: {'on'}
```

Examine the new value for `RemoveProtectionInput`.

```
algParams.RemoveProtectionInput
```

```
ans =

  RemoveProtectionInput with properties:

      Name: 'RemoveProtectionInput'
   Options: {'off'  'on'}
   Primary: 0
     Value: {'off'}
```

Set Algorithm Parameters for `lookup2D` Function Table Entry

Create a code replacement table.

```
crTable = RTW.TflTable;
```

Create a table entry for a function.

```
tableEntry = RTW.TflCFunctionEntry;
```

Identify the table entry as an entry for the `lookup2D` function.

```
setTflCFunctionEntryParameters(tableEntry, ...
    'Key', 'lookup2D', ...
    'Priority', 100, ...
    'ImplementationName', 'myLookup2D');
```

Get the algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry)
```

```
algParams =

  Lookup with properties:

               InterpMethod: [1x1 coder.algorithm.parameter.InterpMethod]
               ExtrapMethod: [1x1 coder.algorithm.parameter.ExtrapMethod]
       UseRowMajorAlgorithm: [1x1 coder.algorithm.parameter.UseRowMajorAlgorithm]
                    RndMeth: [1x1 coder.algorithm.parameter.RndMeth]
          IndexSearchMethod: [1x1 coder.algorithm.parameter.IndexSearchMethod]
          UseLastTableValue: [1x1 coder.algorithm.parameter.UseLastTableValue]
      RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
  SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
    SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
            BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
```

Display the valid values for algorithm parameter `IndexSearchMethod` for the `lookup2D` function.

```
algParams.IndexSearchMethod

ans =

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
    Primary: 0
      Value: {'Binary search'  'Evenly spaced points'  'Linear search'}
```

Set parameter `IndexSearchMethod` to `Evenly spaced points`.

```
algParams.IndexSearchMethod = 'Evenly spaced point';

Error using coder.algorithm.parameter.validateValue (line 58)
Invalid value '{Evenly spaced point}' for algorithm parameter
'coder.algorithm.parameter.IndexSearchMethod'. Valid values are '{Linear
search, Binary search, Evenly spaced points}'.

Error in coder.algorithm.parameter.AlgorithmParameter/set.Value (line 49)
            obj.Value = coder.algorithm.parameter.validateValue(obj, val);

Error in coder.algorithm.parameter.AlgorithmParameter/setAP (line 36)
                obj.Value = value;

Error in coder.algorithm.parameterset.Lookup/set.IndexSearchMethod (line 39)
            obj.IndexSearchMethod = obj.IndexSearchMethod.setAP(value);
```

The code replacement software flags the ‘`s`’ that is missing from ‘`points`’.

Adjust the parameter setting.

```
algParams.IndexSearchMethod = 'Evenly spaced points';
```

Update the parameter settings for the code replacement table entry.

```
setAlgorithmParameters(tableEntry, algParams);
```

Get the updated algorithm parameter settings for the `lookup2D` function table entry.

```
algParams = getAlgorithmParameters(tableEntry);
```

Verify the new value of `IndexSearchMethod`.

```
algParams.IndexSearchMethod

ans =

  IndexSearchMethod with properties:

       Name: 'IndexSearchMethod'
    Options: {'Linear search'  'Binary search'  'Evenly spaced points'}
```

```
    Primary: 0
      Value: {'Evenly spaced points'}
```

Input Arguments

`tableEntry` — Code replacement table entry for a lookup table function

object

Code replacement table entry that you previously created and represents a potential code replacement for a lookup table function. The entry must identify the lookup table function for which you are calling `setAlgorithmParameters`.

1. Create the entry. For example, call the function `RTW.TflCFunctionEntry`.

   ```
   tableEntry = RTW.TflCFunctionEntry;
   ```

2. Specify the name of the lookup table function for which you created the entry. Use the `Key` parameter in a call to `setTflCFunctionEntryParameters`. The following function call specifies the lookup table function `prelookup`.

   ```
   setTflCFunctionEntryParameters(tableEntry, ...
      'Key', 'prelookup', ...
      'Priority', 100, ...
      'ImplementationName', 'Ifx_DpSearch_u8');
   ```

`algParams` — Algorithm parameter settings for a lookup table function

object

Algorithm parameter settings for the lookup table function identified with the `Key` parameter in `tableEntry`.

See Also

`RTW.TflCFunctionEntry` | `RTW.TflTable` | `addEntry` | `getAlgorithmParameters` | `setTflCFunctionEntryParameters`

Topics

“Lookup Table Function Code Replacement”
“Define Code Replacement Mappings”
“Code You Can Replace from MATLAB Code”
“Code You Can Replace From Simulink Models”

Introduced in R2015a

coder.mapping.create

Create C code mapping environment for model

Syntax

```
coder.mapping.create(model)
coder.mapping.create(model,cs)
```

Description

`coder.mapping.create(model)` creates an environment to configure code generation for data and functions of the specified model. Unless you previously opened your model in Code Perspective mode, before calling other default mapping functions, you must call this function.

`coder.mapping.create(model,cs)` creates a code mapping environment for the specified model that includes code customization settings stored in a configuration set object. The configuration set object can specify memory sections for data and functions and a naming rule for shared utilities. Specify a configuration set object to preserve memory section definitions or shared utility naming rules applied to a model in a version of Embedded Coder prior to R2018a.

Examples

Create Environment to Configure Code Mappings for Model

For model `rtwdemo_configdefaults`, create the environment for configuring data and functions for code generation.

```
coder.mapping.create('rtwdemo_configdefaults');
```

After calling this function, use calls to these functions to look up category names, property names, and values that you can use to configure aspects of code generation for model data and functions:

- `coder.mapping.defaults.dataCategories`
- `coder.mapping.defaults.functionCategories`
- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`

Then, specify category, property, and value combinations in calls to `coder.mapping.defaults.set`.

Input Arguments

`model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`cs` — Configuration set object

object

Configuration set object from which to import code customization settings for data and functions.

Example: `'cs_basic'`

Data Types: `char`

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics

“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.mapping.defaults.allowedProperties

Return properties for model default mapping category

Syntax

```
properties = coder.mapping.defaults.allowedProperties(model,
category)
```

Description

`properties = coder.mapping.defaults.allowedProperties(model, category)` returns a cell array of names for properties that are relevant to `category` for the specified model. Use the property names that the `coder.mapping.defaults.allowedProperties` function returns in subsequent calls to `coder.mapping.defaults.allowedValues` and `coder.mapping.defaults.set`.

Examples

Get Properties for Model Default Data Categories

Get a list of the properties for the model default data categories `Inports`, `Outports`, `LocalParameters`, and `InternalData` by using calls to `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Inports')

ans =

  2×1 cell array

    {'StorageClass'}
    {'HeaderFile'  }

coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =
```

```
  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }

coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'LocalParameters')

ans =

  1×1 cell array

    {'StorageClass'}

coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'InternalData')

ans =

  2×1 cell array

    {'StorageClass' }
    {'MemorySection'}
```

Get Properties for Model Default Function Categories

Get a list of the properties for the model default function categories `InitializeTerminate` and `Execution` by using calls to `coder.mapping.defaults.allowedProperties`.

```
catData = coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
  'InitializeTerminate');
catFunctions = coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults',...
'Execution');

catData =

  1×1 cell array

    {'FunctionCustomizationTemplate'}

catFunctions =

  1×1 cell array

    {'FunctionCustomizationTemplate'}
```

Input Arguments

`model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`category` — Name of default mapping category
character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'Inports'`

Data Types: `char`

Output Argument

`properties` — Names of properties for category
cell array

Cell array of names for properties of a default category for the specified model.

See Also

`coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics

“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.mapping.defaults.allowedValues

Return value of property for model default mapping category

Syntax

```
values = coder.mapping.defaults.allowedValues(model,category,
property)
```

Description

`values = coder.mapping.defaults.allowedValues(model,category, property)` returns a cell array of values that are relevant to the specified combination of `category` and `property` for the specified model. To set up category, property, and value combinations for a model, use the value names that the function returns in calls to `coder.mapping.defaults.set`.

Examples

Get Storage Class Values for Default Data Category Local Parameters

Get the list of values that you can specify for property `StorageClass` for model default data category `LocalParameters` by calling `coder.mapping.defaults.allowedValues`.

```
lclparam_scs = coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'LocalParameters',...
  'StorageClass')
lclparam_scs

lclparam_scs =

  15×1 cell array

    {'Default'              }
    {'ExportedGlobal'       }
    {'ImportedExtern'       }
    {'ImportedExternPointer'}
    {'Const'                }
```

```
{'Volatile'             }
{'ConstVolatile'        }
{'Define'               }
{'ImportedDefine'       }
{'ExportToFile'         }
{'ImportFromFile'       }
{'FileScope'            }
{'Struct'               }
{'GetSet'               }
{'CompilerFlag'         }
```

Input Arguments

`model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`category` — Name of default mapping category

character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'LocalParameters'`

Data Types: `char`

`property` — Name of property for default mapping category

character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

Output Argument

`values` — Values for category and property combination
cell array

Cell array of values that are for a default category and property combination for the specified model.

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics

“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.mapping.defaults.dataCategories

Return default mapping categories for model data

Syntax

```
categories = coder.mapping.defaults.dataCategories()
```

Description

`categories = coder.mapping.defaults.dataCategories()` returns a cell array of names for categories of model data elements that you can map to property settings, including a storage class and memory section. The storage class mapped to a category defines how the code generator produces code for that category of data. To set up data category, property, and value combinations for a model, use the category names that the function returns in calls to:

- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`
- `coder.mapping.defaults.set`

Example

Get Model Data Element Categories

Get a list of the available data categories by calling `coder.mapping.defaults.dataCategories`.

```
catData = coder.mapping.defaults.dataCategories()
catData

ans =

  1×8 cell array

  Columns 1 through 4
```

```
    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}

  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

Output Arguments

`categories` — Names of data categories

cell array

Cell array of names for default mapping data categories.

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics

“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.mapping.defaults.functionCategories

Return default mapping categories for model functions

Syntax

```
categories = coder.mapping.defaults.functionCategories()
```

Description

`categories = coder.mapping.defaults.functionCategories()` returns a cell array of names for categories of model functions that you can map to property settings, including a function customization template and memory section. The function customization template mapped to a category defines how the code generator produces code for that category of functions. To set up function category, property, and value combinations for a model, use the category names that the function returns in calls to:

- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`
- `coder.mapping.defaults.set`

Examples

Get Model Function Categories

Get a list of the available function categories by calling `coder.mapping.defaults.functionCategories`.

```
catFunc = coder.mapping.defaults.functionCategories();
catFunc

catFunc =

  1×3 cell array

    {'InitializeTerminate'}    {'Execution'}    {'SharedUtility'}
```

Output Arguments

`categories` — Names of function categories
cell array

Cell array of names for default mapping function categories.

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.get` | `coder.mapping.defaults.set`

Topics

“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.mapping.defaults.get

Return value of property for model default mapping category

Syntax

```
value = coder.mapping.defaults.get(model,category,property)
```

Description

`value = coder.mapping.defaults.get(model,category,property)` returns the value of a property for a data or function default mapping category for a model. To determine valid category and property combinations, use calls to functions `coder.mapping.defaults.dataCategories`, `coder.mapping.defaults.functionCategories`, and `coder.mapping.defaults.allowedProperties`.

Examples

Return Storage Class Setting for Data Imported Into Model

For model `rtwdemo_configuredefaults`, return the storage class that the code generator uses for data imported into the model from external header and definitions files.

Determine the category name to specify for model input data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×8 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}
```

```
  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

Specify `Inports` as the category name.

Identify properties that you can configure for category **Outports** by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =

  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }
```

Use a call to function `coder.mapping.defaults.get` to return the setting for category **Inports** and property `StorageClass`.

```
coder.mapping.defaults.get('rtwdemo_configdefaults', 'Inports', 'StorageClass')

ans =

    'ImportFromFile'
```

Return Memory Section Setting for Model Execution Entry-Point Functions

For model `rtwdemo_configuredefaults`, return the memory section that the code generator uses for model execution entry-point functions, such as `step`.

Determine the category name to specify for execution functions by calling `coder.mapping.defaults.functionCategories`.

```
coder.mapping.defaults.functionCategories()

ans =

  1×3 cell array

    {'InitializeTerminate'}    {'Execution'}    {'SharedUtility'}
```

Specify `Execution` as the category name.

Identify properties that you can configure for category `Execution` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Execution')

ans =

  1×1 cell array

    {'FunctionCustomizationTemplate'}
```

Use a call to function `coder.mapping.defaults.get` to return the setting for category `Execution` and property `FunctionCustomizationTemplate`.

```
coder.mapping.defaults.get('rtwdemo_configdefaults', 'Execution', 'FunctionCustomizationTemplate')

ans =

    'exFastFunction'
```

Input Arguments

`model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`category` — Name of default mapping category

character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'LocalParameters'`

Data Types: `char`

`property` — Name of property for default mapping category

character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

Output Argument

`value` — Value of property for default mapping category

character vector

Character vector that is the setting of the specified default mapping category and property for the specified model.

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.set`

Topics

“Configure Default C Code Generation for Categories of Model Data and Functions”

Introduced in R2018a

coder.mapping.defaults.set

Set value for property of model default mapping category

Syntax

```
coder.mapping.defaults.set(model,category,property,value,...)
```

Description

`coder.mapping.defaults.set(model,category,property,value,...)` sets property values for a data or function default mapping category for a model. To determine valid category, property, and value combinations for a model, use calls to:

- `coder.mapping.defaults.dataCategories`
- `coder.mapping.defaults.functionCategories`
- `coder.mapping.defaults.allowedProperties`
- `coder.mapping.defaults.allowedValues`

Examples

Configure Default Code Generation Settings for Model Output Data

For model `rtwdemo_configuredefaults`, configure how the code generator handles model output data by default.

Determine the category name to specify for model output data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×8 cell array
```

```
  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}

  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

Specify `Outports` as the category name.

Identify properties that you can configure for category `Outports` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'Outports')

ans =

  4×1 cell array

    {'StorageClass'  }
    {'HeaderFile'    }
    {'DefinitionFile'}
    {'Owner'         }
```

For this example, set values for properties `StorageClass`, `HeaderFile`, and `DefinitionFile`.

Look up the values that you can specify for properties `StorageClass`, `HeaderFile`, and `DefinitionFile`.

```
coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'StorageClass')

ans =

  10×1 cell array

    {'Default'              }
    {'ExportedGlobal'       }
    {'ImportedExtern'       }
    {'ImportedExternPointer'}
    {'Volatile'             }
    {'ExportToFile'         }
    {'ImportFromFile'       }
    {'AutoScope'            }
    {'Struct'               }
    {'GetSet'               }

coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'HeaderFile')

ans =

  0×1 empty cell array

coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'Outports', 'DefinitionFile')

ans =
```

```
  0×1 empty cell array
```

Use a call to function `coder.mapping.defaults.set` to configure the default settings. For category `Outports`, set `StorageClass` to `ExportToFile`. Specify `exSysOut.h` and `exSysOut.c` for the header and definition files.

```
coder.mapping.defaults.set('rtwdemo_configdefaults', 'Outports',...
  'Storageclass', 'ExportToFile',...
  'HeaderFile', 'exSysOut.h',...
  'DefinitionFile', 'exSysOut.c')
```

Configure Default Location in Memory for Storing Code Generated for Model Internal Data

For model `rtwdemo_configuredefaults`, configure the default location in memory for storing code generated for model data elements such as signals, states, and zero crossings.

Determine the category name to specify for model internal data by calling `coder.mapping.defaults.dataCategories`.

```
coder.mapping.defaults.dataCategories()

ans =

  1×8 cell array

  Columns 1 through 4

    {'Inports'}    {'Outports'}    {'GlobalParameters'}    {'LocalParameters'}

  Columns 5 through 8

    {'SharedLocalData…'}    {'GlobalDataStores'}    {'InternalData'}    {'Constants'}
```

Specify `InternalData` as the category name.

Identify properties that you can configure for category `InternalData` by calling `coder.mapping.defaults.allowedProperties`.

```
coder.mapping.defaults.allowedProperties('rtwdemo_configdefaults', 'InternalData')

ans =

  2×1 cell array
```

```
    {'StorageClass' }
    {'MemorySection'}
```

To configure the memory location, set the value for property `MemorySection`.

Look up the values that you can specify for property `MemorySection`.

```
coder.mapping.defaults.allowedValues('rtwdemo_configdefaults', 'InternalData', 'MemorySection')

ans =

  5×1 cell array

    {'None'           }
    {'MemVolatile'    }
    {'internalDataMem'}
    {'functionFastMem'}
    {'functionSlowMem'}
```

Use a call to function `coder.mapping.defaults.set` to configure the default setting. For category `InternalData`, set `MemorySection` to `internalDataMem`.

```
coder.mapping.defaults.set('rtwdemo_configdefaults', 'InternalData',...
  'MemorySection', 'internalDataMem')
```

Input Arguments

`model` — Name of model

character vector

Model file, specified as a character vector. The model must be loaded (for example, by using `load_system`) or open. You can omit the `.slx` file extension.

Example: `'myLoadedModel'`

Data Types: `char`

`category` — Name of default mapping category

character vector

Category name, specified as a character vector. To get valid data and function category names, call the functions `coder.mappings.defaults.dataCategories` and `coder.mappings.defaults.functionCategories`.

Example: `'LocalParameters'`

Data Types: `char`

`property` — Name of property for default mapping category
character vector

Property name, specified as a character vector. To get valid property names for a default mapping category, call the function `coder.mappings.defaults.allowedProperties`.

Example: `'StorageClass'`

Data Types: `char`

`value` — Value of property for default mapping category
character vector

Property value, specified as a character vector. To get a list of values that you can specify for a category and property combination, call the function `coder.mappings.defaults.allowedValues`.

Example: `'ExportToFile'`

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues` | `coder.mapping.defaults.dataCategories` | `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get`

Topics

"Configure Default C Code Generation for Categories of Model Data and Functions"

Introduced in R2018a

Functions in Simulink Coder—Alphabetical List

addCompileFlags

Add compiler options to model build information

Syntax

```
addCompileFlags(buildinfo,options,groups)
```

Description

`addCompileFlags(buildinfo,options,groups)` specifies the compiler options to add to the build information.

The function requires the *`buildinfo`* and *`options`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the compiler options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Compiler Flags to OPTS Group

Add the compiler option `-O3` to the build information `myModelBuildInfo` and place the option in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,'-O3','OPTS');
```

Add Compiler Flags to OPT_OPTS Group

Add the compiler options `-Zi` and `-Wall` to the build information `myModelBuildInfo` and place the options in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,'-Zi -Wall','OPT_OPTS');
```

Add Compiler Flags to Build Information

For a non-makefile build environment, add the compiler options `-Zi`, `-Wall`, and `-O3` to the build information `myModelBuildInfo`. Place the options `-Zi` and `-Wall` in the group `Debug` and the option `-O3` in the group `MemOpt`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
  {'Debug' 'MemOpt'});
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`options` — List of compiler options to add to build information

character vector | array of character vectors | string

You can specify the *`options`* argument as a character vector, as an array of character vectors, or as a string You can specify the *`options`* argument as multiple compiler flags within a single character vector, for example `'-Zi -Wall'`. If you specify the *`options`* argument as multiple character vectors, for example, `'-Zi -Wall'` and `'-O3'`, the *`options`* argument is added to the build information as an array of character vectors.

Example: `{'-Zi -Wall' '-O3'}`

`groups` — Optional group name for the added compiler options

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'Debug' 'MemOpt'`, the function relates the *`groups`* to the *`options`* in order of appearance. For example, the *`options`* argument `{'-Zi -Wall' '-O3'}` is an array of character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `'MemOpt'` group.

Note The template makefile-based build process considers only compiler options in the `'OPTS'`, `'OPT_OPTS'`, and `'OPTIMIZATION_FLAGS'` groups when generating the makefile.

Example: `{'Debug' 'MemOpt'}`

See Also

`addDefines` | `addLinkFlags` | `getCompileFlags`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addDefines

Add preprocessor macro definitions to model build information

Syntax

```
addDefines(buildinfo,macrodefs,groups)
```

Description

`addDefines(buildinfo,macrodefs,groups)` specifies the preprocessor macro definitions to add to the build information.

The function requires the *`buildinfo`* and *`macrodefs`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the definitions in a build information object. The function adds definitions to the object based on the order in which you specify them.

Examples

Add Macro Definitions to OPTS Group

Add the macro definition `-DPRODUCTION` to the build information `myModelBuildInfo` and place the definition in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo,'-DPRODUCTION','OPTS');
```

Add Macro Definitions to OPT_OPTS Group

Add the macro definitions `-DPROTO` and `-DDEBUG` to the build information `myModelBuildInfo` and place the definitions in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
  '-DPROTO -DDEBUG','OPT_OPTS');
```

Add Macro Definitions to Build Information

For a non-makefile build environment, add the macro definitions `-DPROTO`, `-DDEBUG`, and `-DPRODUCTION` to the build information `myModelBuildInfo`. Place the definitions `-DPROTO` and `-DDEBUG` in the group `Debug` and the definition `-DPRODUCTION` in the group `Release`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
  {'-DPROTO -DDEBUG' '-DPRODUCTION'}, ...
  {'Debug' 'Release'});
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`macrodefs` — List of macro definitions to add to build information

character vector | array of character vectors | string

You can specify the *`macrodefs`* argument as a character vector, as an array of character vectors, or as a string. You can specify the *`macrodefs`* argument as multiple definitions within a single character vector, for example `'-DRT -DDEBUG'`. If you specify the *`macrodefs`* argument as multiple character vectors, for example`'-DPROTO -DDEBUG'` and `'-DPRODUCTION'`, the *`macrodefs`* argument is added to the build information as an array of character vectors.

Example: `{'-DPROTO -DDEBUG' '-DPRODUCTION'}`

`groups` — Optional group name for the added compiler options

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example `'Debug' 'Release'`, the function relates the *`groups`* to the *`macrodefs`* in order of appearance. For example, the *`macrodefs`* argument `{'-DPROTO -DDEBUG' '-DPRODUCTION'}` is an array of

character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `'Release'` group.

Note The template makefile-based build process considers only macro definitions in the `'OPTS'`, `'OPT_OPTS'`, `'OPTIMIZATION_FLAGS'`, and `'Custom'` groups when generating the makefile.

Example: `{'Debug' 'Release'}`

See Also

`addCompileFlags` | `addLinkFlags` | `getDefines`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addIncludeFiles

Add include files to model build information

Syntax

```
addIncludeFiles(buildinfo,filenames,paths,groups)
```

Description

`addIncludeFiles(buildinfo,filenames,paths,groups)` specifies included files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`paths`* argument to specify the included file paths and use an optional *`groups`* argument to group your options.

The code generator stores the included file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Included File to SysFiles Group

Add the include file `mytypes.h` to the build information `myModelBuildInfo` and place the file in the group `SysFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   'mytypes.h','/proj/src','SysFiles');
```

Add Included Files to AppFiles Group

Add the include files `etc.h` and `etc_private.h` to the build information `myModelBuildInfo`, and place the files in the group `AppFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   {'etc.h' 'etc_private.h'}, ...
   '/proj/src','AppFiles');
```

Add Included Files to SysFiles and AppFiles Groups

Add the include files `etc.h`, `etc_private.h`, and `mytypes.h` to the build information `myModelBuildInfo`. Group the files `etc.h` and `etc_private.h` with the character vector `AppFiles` and the file `mytypes.h` with the character vector `SysFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
{'etc.h' 'etc_private.h' 'mytypes.h'}, ...
   '/proj/src', ...
   {'AppFiles' 'AppFiles' 'SysFiles'});
```

Add Included Files with Wildcard to HFiles Group

Add the include files (`.h` files identified with a wildcard character) in a specified folder to the build information `myModelBuildInfo`, and place the files in the group `HFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   '*.h', '/proj/src', 'HFiles');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of included files to add to build information
character vector | array of character vectors | string

You can specify the *filenames* argument as a character vector, as an array of character vectors, or as a string. If you specify the *filenames* argument as multiple character vectors, for example, `'etc.h' 'etc_private.h'`, the *filenames* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.h'`, and `'*.h*'`.

The function removes duplicate included file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.h'`

paths — List of included file paths to add to build information
character vector | array of character vectors | string

You can specify the *paths* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *paths* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *paths* argument is added to the build information as an array of character vectors.

Example: `'/proj/src'`

groups — Optional group name for the added included files
character vector | array of character vectors | string

You can specify the *groups* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *groups*, for example, `'AppFiles' 'AppFiles' 'SysFiles'`, the function relates the *groups* to the *filenames* in order of appearance. For example, the *filenames* argument `'etc.h' 'etc_private.h' 'mytypes.h'` is an array of character vectors with three elements. The first element is in the `'AppFiles'` group, the second element is in the `'AppFiles'` group, and the third element is in the `'SysFiles'` group.

Example: `'AppFiles' 'AppFiles' 'SysFiles'`

See Also

`addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `findIncludeFiles` | `getIncludeFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addIncludePaths

Add include paths to model build information

Syntax

```
addIncludePaths(buildinfo,paths,groups)
```

Description

`addIncludePaths(buildinfo,paths,groups)` specifies included file paths to add to the build information.

The function requires the *`buildinfo`* and *`paths`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the included file path options in a build information object. The function adds options to the object based on the order in which you specify them.

The code generator does not check whether a specified path is valid.

Examples

Add Include File Path to Build Information

Add the include path `/etcproj/etc/etc_build` to the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
   '/etcproj/etc/etc_build');
```

Add Include File Paths to a Group

Add the include paths `/etcproj/etclib` and `/etcproj/etc/etc_build` to the build information `myModelBuildInfo` and place the files in the group `etc`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
   {'/etcproj/etclib' '/etcproj/etc/etc_build'},'etc');
```

Add Include File Paths to Groups

Add the include paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to the build information `myModelBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/etc_build` with the character vector `etc` and the path `/common/lib` with the character vector `shared`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
   {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
   '/common/lib'}, {'etc' 'etc' 'shared'});
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`paths` — List of included file paths to add to build information

character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

The function removes duplicate include file path entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'/proj/src'`

`groups` — Optional group name for the added included files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'etc' 'etc' 'shared'`, the function relates the *`groups`* to the *`paths`* in order of appearance. For example, the *`paths`* argument `'/etc/proj/etclib' '/etcproj/etc/etc_build' '/common/lib'` is an array of character vectors with three elements. The first element is in the `'etc'` group, the second element is in the `'etc'` group, and the third element is in the `'shared'` group.

Example: `'etc' 'etc' 'shared'`

See Also

`addIncludeFiles` | `addSourceFiles` | `addSourcePaths` | `getIncludePaths` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addLinkFlags

Add link options to model build information

Syntax

```
addLinkFlags(buildinfo,options,groups)
```

Description

`addLinkFlags(buildinfo,options,groups)` specifies the linker options to add to the build information.

The function requires the *`buildinfo`* and *`options`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the linker options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Linker Flags to OPTS Group

Add the linker `-T` option to the build information `myModelBuildInfo` and place the option in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,'-T','OPTS');
```

Add Linker Flags to OPT_OPTS Group

Add the linker options `-MD` and `-Gy` to the build information `myModelBuildInfo` and place the options in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,'-MD -Gy','OPT_OPTS');
```

Add Linker Flags to Build Information

For a non-makefile build environment, add the linker options `-MD`, `-Gy`, and `-T` to the build information `myModelBuildInfo`. Place the options `-MD` and `-Gy` in the group `Debug` and the option `-T` in the group `Temp`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, {'-MD -Gy' '-T'}, ...
   {'Debug' 'Temp'});
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`options` — List of linker options to add to build information

character vector | array of character vectors | string

You can specify the *`options`* argument as a character vector, as an array of character vectors, or as a string. You can specify the *`options`* argument as multiple compiler flags within a single character vector, for example `'-MD -Gy'`. If you specify the *`options`* argument as multiple character vectors, for example, `'-MD -Gy'` and `'-T'`, the *`options`* argument is added to the build information as an array of character vectors.

Example: `{'-MD -Gy' '-T'}`

`groups` — Optional group name for the added linker options

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'Debug' 'Temp'`, the function relates the *`groups`* to the *`options`* in order of appearance. For example, the *`options`* argument `{'-MD -Gy' '-T'}` is an array of character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `Temp'` group.

Example: `{'Debug' 'Temp'}`

See Also

`addCompileFlags` | `addDefines` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addLinkObjects

Add link objects to model build information

Syntax

```
addLinkObjects(buildinfo,linkobjs,paths,priority,precompiled,
linkonly,groups)
```

Description

`addLinkObjects(buildinfo,linkobjs,paths,priority,precompiled, linkonly,groups)` specifies included files and paths to add to the build information.

The function requires the *`buildinfo`*, *`linkobs`*, and *`paths`* arguments. You can optionally select *`priority`* for link objects, select whether the objects are *`precompiled`*, select whether the objects are *`linkonly`* objects, and apply a *`groups`* argument to group your options.

The code generator stores the included link object and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Link Objects to Build Information

Add the linkable objects `libobj1` and `libobj2` to the build information `myModelBuildInfo`. Mark both objects as link-only. Since individual priorities are not specified, the function adds the objects to the vector in the order specified.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo,{'libobj1' 'libobj2'}, ...
```

```
   {'/proj/lib/lib1' '/proj/lib/lib2'},1000, ...
   false,true);
```

Add Prioritized Link-Only Link Objects to Build Information

Add the linkable objects `libobj1` and `libobj2` to the build information `myModelBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Because `libobj2` is assigned the lower numeric priority value and has the higher priority, the function orders the objects such that `libobj2` precedes `libobj1` in the vector.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo, {'libobj1' 'libobj2'}, ...
   {'/proj/lib/lib1' '/proj/lib/lib2'},[26 10]);
```

Add Precompiled Link Objects to MyTest Group

Add the linkable objects `libobj1` and `libobj2` to the build information `myModelBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Mark both objects as precompiled. Group them under the name `MyTest`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo,{'libobj1' 'libobj2'}, ...
   {'/proj/lib/lib1' '/proj/lib/lib2'},[26 10], ...
   true,false,'MyTest');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`linkobjs` — List of linkable object files to add to build information

character vector | array of character vectors | string

You can specify the *`linkobjs`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`linkobjs`* argument as multiple character vectors, for example, `'libobj1' 'libobj2'`, the *`linkobjs`* argument is added to the build information as an array of character vectors.

The function removes duplicate linkable object entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'libobj1'`

`paths` — List of included file paths to add to build information

character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/lib/lib1'` and `'/proj/lib/lib2'`, the *`paths`* argument is added to the build information as an array of character vectors. The number of elements in *`paths`* must match the number of elements in the `linkobjs` argument.

Example: `'/proj/lib/lib1'`

`priority` — List of priority values for link objects to add to build information

`1000` (default) | numeric value | array of numeric values

A numeric value or an array of numeric values that indicates the relative priority of each specified link object. Lower values have higher priority.

Example: `1000`

`precompiled` — List of precompiled indicators for link objects to add to build information

false (default) | true | array of logical values

A logical value or an array of logical values that indicates whether each specified link object is precompiled. The logical value `true` indicates precompiled.

Example: `false`

`linkonly` — List of link-only indicators for link objects to add to build information

`false` (default) | true

A logical value or an array of logical values that indicates whether each specified link object is link-only (not precompiled). The logical value `true` indicates link-only. If *`linkonly`* is `true`, the value of the *`precompiled`* argument is ignored.

Example: `false`

`groups` — Optional group name for the added link object files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'MyTest1' 'MyTest2'`, the function relates the *`groups`* to the *`linkobjs`* in order of appearance. For example, the *`linkobjs`* argument `'libobj1' 'libobj2'` is an array of character vectors with two elements. The first element is in the `'MyTest1'` group, and the second element is in the `'MyTest2'` group.

Example: `'MyTest1' 'MyTest2'`

See Also

`addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `findIncludeFiles` | `getIncludeFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addNonBuildFiles

Add nonbuild-related files to model build information

Syntax

```
addNonBuildFiles(buildinfo,filenames,paths,groups)
```

Description

`addNonBuildFiles(buildinfo,filenames,paths,groups)` specifies nonbuild-related files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`paths`* argument to specify the included file paths and use an optional *`groups`* argument to group your options.

The code generator stores the nonbuild-related file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Nonbuild File to DocFiles Group

Add the nonbuild-related file `readme.txt` to the build information `myModelBuildInfo`, and place the file in the group `DocFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
   'readme.txt','/proj/docs','DocFiles');
```

Add Nonbuild Files to DLLFiles Group

Add the nonbuild-related files `myutility1.dll` and `myutility2.dll` to the build information `myModelBuildInfo`, and place the files in the group `DLLFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
   {'myutility1.dll' 'myutility2.dll'}, ...
   '/proj/dlls','DLLFiles');
```

Add Nonbuild Files with Wildcard to DLLFiles Group

Add nonbuild-related files (`.dll` files identified with a wildcard character) in a specified folder to the build information `myModelBuildInfo`, and place the files in the group `DLLFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
   '*.dll','/proj/dlls','DLLFiles');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of nonbuild-related files to add to build information
character vector | array of character vectors | string

You can specify the *`filenames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`filenames`* argument as multiple character vectors, for example, `'etc.dll' 'etc_private.dll'`, the *`filenames`* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.dll'`, and `'*.d*'`.

The function removes duplicate nonbuild-related file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.dll'`

`paths` — List of nonbuild-related file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/dll'` and `'/proj/docs'`, the *`paths`* argument is added to the build information as an array of character vectors.

Example: `'/proj/dll'`

`groups` — Optional group name for the added nonbuild-related files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'DLLFiles' 'DLLFiles' 'DocFiles'`, the function relates the *`groups`* to the *`filenames`* in order of appearance. For example, the *`filenames`* argument `''myutility1.dll'' ''myutility2.dll'' 'readme.txt'` is an array of character vectors with three elements. The first element is in the `'DLLFiles'` group, the second element is in the `'DLLFiles'` group, and the third element is in the `'DocFiles'` group.

Example: `'DLLFiles' 'DLLFiles' 'DocFiles'`

See Also

`getNonBuildFiles`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2008a

addSourceFiles

Add source files to model build information

Syntax

```
addSourceFiles(buildinfo,filenames,paths,groups)
```

Description

`addSourceFiles(buildinfo,filenames,paths,groups)` specifies source files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the source file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Source File to Drivers Group

Add the source file `driver.c` to the build information `myModelBuildInfo` and place the file in the group `Drivers`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo,'driver.c', ...
   '/proj/src', 'Drivers');
```

Add Source Files to a Group

Add the source files `test1.c` and `test2.c` to the build information `myModelBuildInfo` and place the files in the group `Tests`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
   {'test1.c' 'test2.c'}, ...
   '/proj/src','Tests');
```

Add Source Files to Groups

Add the source files `test1.c`, `test2.c`, and `driver.c` to the build information `myModelBuildInfo`. Group the files `test1.c` and `test2.c` with the character vector `Tests`. Group the file `driver.c` with the character vector `Drivers`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
   {'test1.c' 'test2.c' 'driver.c'}, ...
   '/proj/src', ...
   {'Tests' 'Tests' 'Drivers'});
```

Add Source Files with Wildcard to CFiles Group

Add the `.c` files in a specified folder to the build information `myModelBuildInfo` and place the files in the group `CFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
   '*.c','/proj/src','CFiles');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of source files to add to build information
character vector | array of character vectors | string

You can specify the *`filenames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`filenames`* argument as multiple character vectors, for example, `'etc.c' 'etc_private.c'`, the *`filenames`* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.c'`, and `'*.c*'`.

The function removes duplicate included file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.c'`

`paths` — List of source file paths to add to build information

character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

Example: `'/proj/src'`

`groups` — Optional group name for the added source files

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'Tests' 'Tests' 'Drivers'`, the function relates the *`groups`* to the *`filenames`* in order of appearance. For example, the *`filenames`* argument `'test1.c' 'test2.c' 'driver.c'` is an array of character vectors with three elements. The first element is in the `'Tests'` group, and the second element is in the `'Tests'` group, and the third element is in the `'Drivers'` group.

Example: `'Tests' 'Tests' 'Drivers'`

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourcePaths` | `getSourceFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addSourcePaths

Add source paths to model build information

Syntax

```
addSourcePaths(buildinfo,paths,groups)
```

Description

`addSourcePaths(buildinfo,paths,groups)` specifies source file paths to add to the build information.

The function requires the *`buildinfo`* and *`paths`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the source file path options in a build information object. The function adds options to the object based on the order in which you specify them.

The code generator does not check whether a specified path is valid.

Note If you want to add source files and the corresponding file paths to model build information, use the `addSourceFiles` function. Do not use `addSourcePaths`.

Examples

Add Source File Path to Build Information

Add the source path `/etcproj/etc/etc_build` to the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, ...
   '/etcproj/etc/etc_build');
```

Add Source File Paths to a Group

Add the source paths `/etcproj/etclib` and `/etcproj/etc/etc_build` to the build information `myModelBuildInfo` and place the files in the group `etc`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, ...
   {'/etcproj/etclib' '/etcproj/etc/etc_build'},'etc');
```

Add Source File Paths to Groups

Add the source paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to the build information `myModelBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/etc_build` with the character vector `etc` and the path `/common/lib` with the character vector `shared`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,...
   {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
   '/common/lib'}, {'etc' 'etc' 'shared'});
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`paths` — List of source file paths to add to build information

character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

The function removes duplicate source file path entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'/proj/src'`

`groups` — Optional group name for the added source files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'etc' 'etc' 'shared'`, the function relates the *`groups`* to the *`paths`* in order of appearance. For example, the *`paths`* argument `'/etc/proj/etclib' '/etcproj/etc/etc_build' '/common/lib'` is an array of character vectors with three elements. The first element is in the `'etc'` group, the second element is in the `'etc'` group, and the third element is in the `'shared'` group.

Example: `'etc' 'etc' 'shared'`

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `getSourcePaths` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

addTMFTokens

Add template makefile (TMF) tokens to model build information

Syntax

```
addTMFTokens(buildinfo,tokennames,tokenvalues,groups)
```

Description

`addTMFTokens(buildinfo,tokennames,tokenvalues,groups)` specifies TMF tokens and values to add to the build information.

To provide build-time information to help customize makefile generation, call the `addTMFTokens` function inside a post-code-generation command. The tokens specified in the `addTMFTokens` function call must be handled in the template makefile (TMF) for the target selected for your model. For example, you can call `addTMFTokens` in a post-code-generation command to add a TMF token named `|>CUSTOM_OUTNAME<|` with a token value that specifies an output file name for the build. To achieve the result you want, the TMF must apply an action with the value of `|>CUSTOM_OUTNAME<|`. (See “Examples” on page 2-0 .)

The `addTMFTokens` function adds specified TMF token names and values to the model build information. The code generator stores the TMF tokens in a vector. The function adds the tokens to the end of the vector in the order that you specify them.

The function requires the *`buildinfo`*, *`tokennames`*, and *`tokenvalues`* arguments. You can use an optional *`groups`* argument to group your options. You can specify *`groups`* as a character vector or as an array of character vectors.

Examples

Add TMF Tokens to Build Information

Inside a post-code-generation command, add the TMF token `|>CUSTOM_OUTNAME<|` and its value to build information `myModelBuildInfo`, and place the token in the group `LINK_INFO`.

```
myModelBuildInfo = RTW.BuildInfo;
addTMFTokens(myModelBuildInfo, ...
             '|>CUSTOM_OUTNAME<|', 'foo.exe', 'LINK_INFO');
```

Apply Build Information as Tokens in TMF Build

In the TMF for the target selected for your model, this code uses the token value to achieve the result that you want:

```
CUSTOM_OUTNAME = |>CUSTOM_OUTNAME<|
...
target:
$(LD) -o $(CUSTOM_OUTNAME) ...
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`tokennames` — Specifies names of TMF tokens to add to the build information
character vector | array of character vectors | string

You can specify the *`tokennames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`tokennames`* argument as multiple character vectors, for example, `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'`, the *`tokennames`* argument is added to the build information as an array of character vectors.

Example: `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'`

`tokenvalues` — Specifies TMF token values (for the added tokens) to add to the build information
character vector | array of character vectors | string

You can specify the *`tokenvalues`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`tokenvalues`* argument as multiple

character vectors, for example, `'|>CUSTOM_OUTNAME<|' 'PCWIN64'`, the *`tokennames`* argument is added to the build information as an array of character vectors.

Example: `'foo.exe' 'PCWIN64'`

`groups` — Optional group name for the added TMF tokens
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'LINK_INFO' 'COMPUTER_INFO'`, the function relates the *`groups`* to the *`tokennames`* in order of appearance. For example, the *`tokennames`* argument `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'` is an array of character vectors with two elements. The first element is in the `'LINK_INFO'` group, and the second element is in the `'COMPUTER_INFO'` group.

Example: `'LINK_INFO' 'COMPUTER_INFO'`

See Also

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2009b

buildStandaloneCoderAssumptions

Create application to check code generator assumptions

Syntax

```
buildStandaloneCoderAssumptions(buildFolder)
```

Description

`buildStandaloneCoderAssumptions(buildFolder)` creates an application for your target hardware to check code generator assumptions. The application checks that code generator assumptions based on model parameter settings or build configuration settings are correct with reference to the target hardware.

The function creates the target application in the `buildFolder\coderassumptions\standalone` subfolder.

Examples

Create Application to Check Code Generator Assumptions

For an example that shows how to create an application to check code generator assumptions, see “Check Code Generator Assumptions for Development Computer”.

Input Arguments

`buildFolder` — Build folder
character vector | string scalar

Path to the build folder that contains the generated code.

See Also

Topics

“Check Code Generation Assumptions”

Introduced in R2018b

coder.buildstatus.close

Close build process status window

Syntax

```
coder.buildstatus.close()
coder.buildstatus.close(model)
coder.buildstatus.close(subsystem)
```

Description

`coder.buildstatus.close()` closes open **Build Process Status** windows.

The **Build Process Status** window supports parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

`coder.buildstatus.close(model)` closes the **Build Process Status** window for the `model`.

`coder.buildstatus.close(subsystem)` closes the **Build Process Status** window for the `subsystem`.

Examples

Close Build Process Status Windows

Close open **Build Process Status** windows.

```
coder.buildstatus.close()
```

Close Build Process Status Window for a Model

After generating code for `rtwdemo_counter`, close the **Build Process Status** window for the model.

```
coder.buildstatus.close('rtwdemo_counter')
```

Close Build Process Status Window for a Subsystem

Close the **Build Process Status** window for the subsystem 'Amplifier' in model 'rtwdemo_counter'.

```
coder.buildstatus.close('rtwdemo_counter/Amplifier')
```

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: 'rtwdemo_counter'

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: 'rtwdemo_counter/Amplifier'

Data Types: `char`

See Also

`coder.buildstatus.open` | `coder.report.close` | `rtwbuild` | `slbuild`

Topics

"View Build Process Status" (Simulink Coder)

Introduced in R2018a

coder.buildstatus.open

Open build process status window

Syntax

```
coder.buildstatus.open(model)
coder.buildstatus.open(model,systemTarget)
```

Description

`coder.buildstatus.open(model)` opens the **Build Process Status** window for the `model`.

The **Build Process Status** window supports parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

If the current working folder is the model build folder and the folder contains information from a previous parallel build, opening the **Build Process Status** window displays the previous build information. When you start a model parallel build, the current build information replaces the previous build information in the window.

`coder.buildstatus.open(model,systemTarget)` opens the **Build Process Status** window for the `model` and displays the `model` tab. The available tabs are **Simulation Targets** and **Code Generation Targets**

Examples

Open Build Process Status Window for a Model

After generating code for model `'rtwdemo_parabuild_a_1'`, open the **Build Process Status** window for the model.

```
addpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','rtwdemo_parallelbuild'))
coder.buildstatus.open('rtwdemo_parabuild_a_1')
```

Open Build Process Status Window with Simulation Targets

Open the **Build Process Status** window for the model `'rtwdemo_parabuild_a_1'` and display the **Simulation Targets** tab.

```
addpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','rtwdemo_parallelbuild'))
coder.buildstatus.open('rtwdemo_parabuild_a_1','sim')
```

Input Arguments

`model` — Model name

character vector | string scalar

Model name specified as a character vector or a string scalar

Example: `'rtwdemo_parabuild_a_1'`

Data Types: `char` | `string`

`systemTarget` — System targets name

`sim` | `rtw`

System targets tab name specified as a character vector or string scalar, `sim` for **Simulation Targets** and `rtw` for **Code Generation Targets**. When build information is available for a system target from a previous build, the *`systemTarget`* argument directs the **Build Status** dialog box to display the tab for the system target. If this optional argument is omitted, when build information is available, dialog opens both the **Simulation Targets** tab and **Code Generation Targets** tab. If build information for a target is not available, the dialog does not open the corresponding system targets tab.

Example: `'rtw'`

Data Types: `char` | `string`

See Also

`coder.buildstatus.close` | `coder.report.open` | `rtwbuild` | `slbuild`

Topics

“View Build Process Status” (Simulink Coder)

Introduced in R2018a

coder.codedescriptor.CodeDescriptor class

Package: coder.codedescriptor

Return information about generated code

Description

Create a `coder.codedescriptor.CodeDescriptor` object to access all the methods defined within the code descriptor API. The `coder.codedescriptor.CodeDescriptor` object describes the data interfaces, function interfaces, global data stores, local and global parameters in the generated code.

Construction

`codeDescObj = coder.getCodeDescriptor(model)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified model.

`codeDescObj = coder.getCodeDescriptor(folder)` creates a `coder.codedescriptor.CodeDescriptor` object for the model in the build folder specified in `folder`.

Properties

`ModelName` — Name of the model
character vector (default)

Name of the model for which the code descriptor object is invoked.

Example: `'rtwdemo_comments'`

`BuildFolder` — Build folder
character vector (default)

Path of the build folder where the model is built.

Example: `'C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw'`

Methods

getAllDataInterfaceTypes	Return data interface types
getAllFunctionInterfaceTypes	Return function interface types
getArrayLayout	Return array layout of the generated code
getDataInterfaces	Return information of the specified data interface
getDataInterfaceTypes	Return data interface types in the generated code
getFunctionInterfaces	Return information of the specified function interface
getFunctionInterfaceTypes	Return function interface types in the generated code
getReferencedModelCodeDescriptor	Return `coder.codedescriptor.CodeDescriptor` object for the specified referenced model
getReferencedModelNames	Return names of the referenced models

Example

Create a `coder.codedescriptor.CodeDescriptor` object for the required model that is built.

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```

   The `codeDescObj` with these properties is created:

   ```
       ModelName: 'rtwdemo_comments'
        BuildDir: 'C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw'
   ```

3. Return a list of all available function interface types.

   ```
   allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
   ```

   `allFunctionInterfaceTypes` has these values:

```
{'Initialize'}
{'Output'    }
{'Update'    }
{'Terminate' }
```

See Also

`getCodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getAllDataInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return data interface types

Syntax

```
allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)
```

Description

`allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)` returns a list of the data interface types. This list is not specific to any model.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`allDataInterfaceTypes` — Data interface types available
cell array of character vectors

A list of available data interface types.

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the required model that is built, then list the available data interface types.

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```

3. Return a list of available data interface types.

   ```
   allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)
   ```

   `allDataInterfaceTypes` has these values:

   ```
   {'Inports'         }
   {'Outports'        }
   {'Parameters'      }
   {'GlobalDataStores'}
   {'GlobalParameters'}
   {'LocalParameters' }
   {'InternalData'    }
   ```

   In a model, there can be `GlobalParameters` and/or `LocalParameters`. The data interface type `Parameters` consist of a consolidated list of both types of parameters.

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `getDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getAllFunctionInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return function interface types

Syntax

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(
codeDescObj)
```

Description

`allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)` returns a list of the function interface types. The returned list is not specific to any model.

Input Arguments

codeDescObj — Code Descriptor object

`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

allFunctionInterfaceTypes — Function interface types available

cell array of character vectors

A list of the available function interface types.

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the required model which is built, then list the available function interface types.

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```

3. Return a list of available function interface types.

   ```
   allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
   ```

   `allFunctionInterfaceTypes` has these values:

   ```
   {'Initialize'}
   {'Output'    }
   {'Update'    }
   {'Terminate' }
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `getFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code” (Simulink Coder)
“Configure Code Generation for Model Entry-Point Functions” (Simulink Coder)

Introduced in R2018a

getArrayLayout

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return array layout of the generated code

Syntax

```
arrayLayout = getArrayLayout(codeDescObj)
```

Description

`arrayLayout = getArrayLayout(codeDescObj)` returns the array layout of the model for which the code is generated.

Input Arguments

`codeDescObj` — Code Descriptor object

`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`arrayLayout` — Array layout of the generated code

character vectors

Array layout specified for the model by using the model configuration parameter **Array layout** (Simulink Coder).

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the model that is built, then list the array layout of the generated code.

1. Open a model.

   ```
   rtwdemo_comments
   ```
2. Specify the model configuration parameter **Array layout** as `Row-major`. Alternatively, in the command window, use these commands:

   ```
   set_param('rtwdemo_comments', 'ArrayLayout','Row-major');
   ```
3. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```
4. Create a `coder.codedescriptor.CodeDescriptor` object for the model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```
5. Return the array layout of the generated code.

   ```
   arrayLayout = getArrayLayout(codeDescObj)
   ```

   `arrayLayout` has this value:

   ```
   'Row-major'
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code” (Simulink Coder)
“Code Generation of Matrices and Arrays” (Simulink Coder)

Introduced in R2018b

getDataInterfaces

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return information of the specified data interface

Syntax

```
dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)
```

Description

`dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)` returns the type of data, SID, graphical name, timing, implementation, and variant information on the data interface that `dataInterfaceName` specifies.

Input Arguments

`codeDescObj` — Code Descriptor object

`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

`dataInterfaceName` — Name of data interface

`Inports` | `Outports` | `Parameters` | `GlobalDataStores` | `GlobalParameters` | `LocalParameters` | `InternalData`

`dataInterfaceName` specifies the name of a data interface. To get a list of all the data interfaces in the generated code, call `getDataInterfaceTypes()`.

Data Types: `string`

Output Arguments

`dataInterface` — `coder.descriptor.DataInterface` object with properties of specified data interface type
`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

The `coder.descriptor.DataInterface` object describes information about the specified data interface such as type of data, SID, graphical name, timing, implementation, and variant information.

Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```
3. Return a list of all data interface types in the generated code.

   ```
   dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
   ```

   `dataInterfaceTypes` has these values:

   ```
       {'Inports'         }
       {'Outports'        }
       {'Parameters'      }
       {'GlobalParameters'}
   ```
4. Return properties of Inport blocks in the generated code.

   ```
   dataInterface = getDataInterfaces(codeDescObj, 'Inports')
   ```

   `dataInterface` is an array of `coder.descriptor.DataInterface` objects. Obtain the details of the first Inport block of the model by accessing the first location in the array.

   ```
   dataInterface(1)
   ```

   The first `coder.descriptor.DataInterface` object with properties is returned.

   ```
                Type: [1×1 coder.descriptor.types.Double]
                 SID: 'rtwdemo_comments:1'
   ```

```
     GraphicalName: 'In1'
       VariantInfo: [0×0 coder.descriptor.VariantInfo]
    Implementation: [1×1 coder.descriptor.StructExpression]
            Timing: [1×1 coder.descriptor.TimingInterface]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaceTypes` | `coder.descriptor.DataInterface`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getDataInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return data interface types in the generated code

Syntax

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

Description

`dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)` returns a list of the data interface types in the generated code. To get a list of the available data interfaces, call `getAllDataInterfaceTypes()`.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`dataInterfaceTypes` — Data interface types in the generated code
cell array of character vectors

A list of the data interface types in the generated code.

Examples

1. Build the model.

```
rtwbuild('rtwdemo_counter')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
```

3. Return a list of data interface types in the generated code.

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

`dataInterfaceTypes` has these values for model `rtwdemo_counter`:

```
    {'Inports'     }
    {'Outports'    }
    {'InternalData'    }
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getFunctionInterfaces

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return information of the specified function interface

Syntax

```
functionInterface = getFunctionInterfaces(codeDescObj,
functionInterfaceName)
```

Description

`functionInterface = getFunctionInterfaces(codeDescObj, functionInterfaceName)` returns the function prototype, input arguments, return arguments, variant conditions, and timing information of the function interface that `functionInterfaceName` specifies.

Input Arguments

`codeDescObj` — Code Descriptor object

`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

`functionInterfaceName` — Name of function interface

`Initialize` | `Output` | `Update` | `Terminate`

`functionInterfaceName` specifies the name of a function interface. A list of all the function interfaces in the generated code is returned by `getFunctionInterfaceTypes()`.

Data Types: `string`

Output Arguments

`functionInterface` — `coder.descriptor.FunctionInterface` object with properties of specified function interface type
`coder.descriptor.FunctionInterface` object | array of `coder.descriptor.FunctionInterface` objects

The `coder.descriptor.FunctionInterface` object describes information about the specified function interface such as function prototype, input arguments, return arguments, variant conditions, and timing information.

Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```

3. Return a list of all function interface types in the generated code.

   ```
   functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
   ```

   These are the function interface types in the generated code of model `rtwdemo_comments`:

   ```
       {'Initialize'}
       {'Output'    }
   ```

4. Return properties of a specified function interface in the generated code.

   ```
   functionInterface = getFunctionInterfaces(codeDescObj, 'Output')
   ```

   `functionInterface` is a `coder.descriptor.FunctionInterface` object.

   ```
              Prototype: [1×1 coder.descriptor.types.Prototype]
         ActualReturn: [0×0 coder.descriptor.DataInterface]
          VariantInfo: [0×0 coder.descriptor.VariantInfo]
               Timing: [1×1 coder.descriptor.TimingInterface]
           ActualArgs: [1x0 coder.descriptor.DataInterface List]
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaceTypes` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getFunctionInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return function interface types in the generated code

Syntax

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
```

Description

`functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)` returns a list of the function interface types in the generated code. To get a list of the available function interfaces, call `getAllFunctionInterfaceTypes()`.

Input Arguments

`codeDescObj` — Code Descriptor object

`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`functionInterfaceTypes` — Function interface types in the generated code

cell array of character vectors

A list of the data interface types in the generated code.

Examples

1. Build the model.

```
rtwbuild('rtwdemo_counter')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
```

3. Return a list of function interface types in the generated code.

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
```

`functionInterfaceTypes` has these values for model `rtwdemo_counter`:

```
{'Output'    }
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getReferencedModelCodeDescriptor

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return `coder.codedescriptor.CodeDescriptor` object for the specified referenced model

Syntax

```
refCodeDescriptor = getReferencedModelCodeDescriptor(codeDescObj,
refModelName)
```

Description

`refCodeDescriptor = getReferencedModelCodeDescriptor(codeDescObj, refModelName)` returns the `coder.codedescriptor.CodeDescriptor` object for the referenced model specified in `refModelName`.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

`refModelName` — Name of referenced model
string

`refModelName` can take any name from the list of referenced models returned by `getReferencedModelNames()`.

Output Arguments

`refCodeDescriptor` — `coder.codedescriptor.CodeDescriptor` object for the specified referenced model
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for the specified referenced model.

Examples

1. Build the model.

```
rtwbuild('rtwdemo_async_mdlreftop')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_async_mdlreftop')
```

3. Return a list of referenced models.

```
refModels = getReferencedModelNames(codeDescObj)
```

`refModels` contains the list of referenced models for `rtwdemo_async_mdlreftop`.

```
{'rtwdemo_async_mdlrefbot'}
```

Obtain the `coder.codedescriptor.CodeDescriptor` object for any of the referenced models.

```
refCodeDescriptorObj = getReferencedModelCodeDescriptor(codeDescObj, 'rtwdemo_async_mdlrefbot')
```

`refCodeDescriptorObj` is the `coder.codedescriptor.CodeDescriptor` object for `rtwdemo_async_mdlrefbot` model.

```
    ModelName: 'rtwdemo_async_mdlrefbot'
     BuildDir: 'C:\Users\Desktop\Work\slprj\tornado\rtwdemo_async_mdlrefbot'
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelNames` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getReferencedModelNames

Class: coder.codedescriptor.CodeDescriptor
Package: coder.codedescriptor

Return names of the referenced models

Syntax

```
refModels = getReferencedModelNames(codeDescObj)
```

Description

`refModels = getReferencedModelNames(codeDescObj)` returns a list of referenced models for a `coder.codedescriptor.CodeDescriptor` object.

Input Arguments

`codeDescObj` — Code Descriptor object
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about generated code.

Output Arguments

`refModels` — Names of referenced models
cell array of character vectors

A list of referenced models.

Examples

1. Build the model.

```
rtwbuild('rtwdemo_async_mdlreftop')
```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_async_mdlreftop')
```

3. Return a list of referenced models.

```
refModels = getReferencedModelNames(codeDescObj)
```

`refModels` has the list of referenced models.

```
{'rtwdemo_async_mdlrefbot'}
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelCodeDescriptor`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

coder.descriptor.DataInterface class

Package: coder.descriptor

Return information about different types of data interfaces

Description

The `coder.descriptor.DataInterface` object describes various properties for a specified data interface in the generated code. These are the different types of data interfaces:

- Root-level inports and outports: An interface between the model and external models or systems, for exchanging data.
- Block-specific parameters: Local and Global parameters that describes the data for the block.
- Global Data Store: A repository to store global data that can be written and read.
- Internal data: Internal data structures include DWork vectors, block I/O, zero-crossings.

Construction

`dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.DataInterface` object. `codeDescObj` is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

Input Arguments

`dataInterfaceName` — Name of data interface
`Inports` | `Outports` | `Parameters` | `GlobalDataStores` | `GlobalParameters` | `LocalParameters` | `InternalData`

Name of the specified data interface.

Example: `'Inports'`

Data Types: `string`

Properties

`Type` — Type of data
`coder.descriptor.types` object

The data type associated with the data such as `integer`, `double`, `matrix`, and its properties.

`SID` — Simulink identifier
character vector

The Simulink identifier (SID) is a unique number within the model that Simulink assigns to the block.

`GraphicalName` — Name of graphical entity
character vector

The name of the associated graphical entity.

`VariantInfo` — Variant conditions in the model
`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the data interface.

`Implementation` — Description of implementation of data
`coder.descriptor.DataImplementation` object

The description of how the data in the generated code is implemented. This property describes characteristics such as data type and size. In addition, it describes how the data is accessed or declared in the code. The property describes if the data is declared as a variable or structure member.

`Timing` — Data access rate in run-time environment
`coder.descriptor.TimingInterface` object

The rate at which data is accessed in a run-time environment.

Limitations

A bitfield data structure is generated if you enable these configuration parameters:

- **Pack Boolean data into bitfields**
- **Use bitset for storing state configuration**
- **Use bitset for storing Boolean data**

If the `coder.descriptor.DataInterface` represents a bitfield data structure, the Implementation property of the `coder.descriptor.DataInterface` object is empty.

Example

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```

3. Return a list of all data interface types in the generated code.

   ```
   dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
   ```

   `dataInterfaceTypes` has these values:

   ```
       {'Inports'         }
       {'Outports'        }
       {'Parameters'      }
       {'GlobalParameters'}
       {'InternalData'    }
   ```

4. Return properties of a specified data interface in the generated code.

   ```
   dataInterface = getDataInterfaces(codeDescObj, 'Inports')
   ```

   `dataInterface` is an array of `coder.descriptor.DataInterface` objects. Obtain the details of the first Inport block of the model by accessing the first location in the array.

   ```
   dataInterface(1)
   ```

   The first `coder.descriptor.DataInterface` object with properties is returned.

```
             Type: [1×1 coder.descriptor.types.Double]
              SID: 'rtwdemo_comments:1'
      GraphicalName: 'In1'
        VariantInfo: [0×0 coder.descriptor.VariantInfo]
     Implementation: [1×1 coder.descriptor.StructExpression]
             Timing: [1×1 coder.descriptor.TimingInterface]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaceTypes` | `getDataInterfaces`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

coder.descriptor.FunctionInterface class

Package: coder.descriptor

Return information about entry-point functions

Description

The function interfaces are the entry-point functions in the generated code. The `coder.descriptor.FunctionInterface` object describes various properties for a specified function interface. The different types of function interfaces are:

- Initialize: Contains initialization code for the model and is called once at the start of your application code. See `model_initialize`.
- Output: Contains the output code for the blocks in the model. See `model_step`.
- Update: Contains the update code for the blocks in the model. See `model_step`.
- Terminate: Contains the termination code for the model and is called as part of a system shutdown. See `model_terminate`.

Construction

`functionInterface = getFunctionInterfaces(codeDescObj, functionInterfaceName)` creates a `coder.descriptor.FunctionInterface` object. `codeDescObj` is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

Input Arguments

`functionInterfaceName` — Name of function interface
`Initialize` | `Output` | `Update` | `Terminate`

Name of the specified function interface

Example: `'Output'`

Data Types: `string`

Properties

`Prototype` — Description of function prototype
`coder.descriptor.types` object

The description of the function prototype including function return value, name, arguments, header, and source files.

`ActualReturn` — Return arguments from the function
`coder.descriptor.DataInterface` object

The data that the function returns as a return argument. When there is no data returned from the function, this field is empty.

`VariantInfo` — Variant conditions in the model
`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the function interface.

`Timing` — Function access rate in run-time environment
`coder.descriptor.TimingInterface` object

The rate at which function is accessed in a run-time environment.

`ActualArgs` — Input arguments to the function
`coder.descriptor.DataInterfaceList` object

The data passed as arguments to the function. When there is no data passed as an argument to the function, this field is empty.

Example

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```
3. Return a list of all function interface types in the generated code.

   ```
   functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
   ```

`functionInterfaceTypes` consists this:

```
    {'Initialize'}
    {'Output'    }
```

4 Return properties of a specified function interface in the generated code.

```
functionInterface = getFunctionInterfaces(codeDescObj, 'Output')
```

`functionInterface` is a `coder.descriptor.FunctionInterface` object.

```
        Prototype: [1×1 coder.descriptor.types.Prototype]
     ActualReturn: [0×0 coder.descriptor.DataInterface]
      VariantInfo: [0×0 coder.descriptor.VariantInfo]
           Timing: [1×1 coder.descriptor.TimingInterface]
       ActualArgs: [1x0 coder.descriptor.DataInterface List]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaceTypes` | `getFunctionInterfaces`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

coder.report.close

Close HTML code generation report

Syntax

```
coder.report.close()
```

Description

`coder.report.close()` closes the HTML code generation report.

Examples

Close code generation report for a model

After opening a code generation report for `rtwdemo_counter`, close the report.

```
coder.report.close()
```

See Also

`coder.report.generate` | `coder.report.open`

Topics

"Reports for Code Generation" (Simulink Coder)

Introduced in R2012a

coder.report.generate

Generate HTML code generation report

Syntax

```
coder.report.generate(model)
coder.report.generate(subsystem)
coder.report.generate(model,Name,Value)
```

Description

`coder.report.generate(model)` generates a code generation report for the `model`. The build folder for the model must be present in the current working folder.

`coder.report.generate(subsystem)` generates the code generation report for the `subsystem`. The build folder for the subsystem must be present in the current working folder.

`coder.report.generate(model,Name,Value)` generates the code generation report using the current model configuration and additional options specified by one or more `Name,Value` pair arguments. Possible values for the `Name,Value` arguments are parameters on the **Code Generation > Report** pane. Without modifying the model configuration, using the `Name,Value` arguments you can generate a report with a different report configuration.

Examples

Generate Code Generation Report for Model

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Build the model. The model is configured to create and open a code generation report.

```
rtwbuild('rtwdemo_counter');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report.

```
coder.report.generate('rtwdemo_counter');
```

Generate Code Generation Report for Subsystem

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Build the subsystem. The model is configured to create and open a code generation report.

```
rtwbuild('rtwdemo_counter/Amplifier');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report for the subsystem.

```
coder.report.generate('rtwdemo_counter/Amplifier');
```

Generate Code Generation Report to Include Static Code Metrics Report

Generate a code generation report to include a static code metrics report after the build process, without modifying the model.

Open the model `rtwdemo_hyperlinks`.

```
open rtwdemo_hyperlinks
```

Build the model. The model is configured to create and open a code generation report.

```
rtwbuild('rtwdemo_hyperlinks');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report that includes the static code metrics report.

```
coder.report.generate('rtwdemo_hyperlinks',
'GenerateCodeMetricsReport','on');
```

The code generation report opens. In the left navigation pane, click **Static Code Metrics Report** to view the report.

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: ‘`rtwdemo_counter`’

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: ‘`rtwdemo_counter/Amplifier`’

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Each `Name,Value` argument corresponds to a parameter on the Configuration Parameters **Code Generation > Report** pane. When the configuration parameter `GenerateReport` is `on`, the parameters are enabled. The `Name,Value` arguments are

used only for generating the current report. The arguments will override, but not modify, the parameters in the model configuration. The following parameters require an Embedded Coder license.

Example: `'GenerateWebview','on','GenerateCodeMetricsReport','on'` includes a model Web view and static code metrics in the code generation report.

Navigation

`IncludeHyperlinkInReport` — Code-to-model hyperlinks

`'off'` | `'on'`

Code-to-model hyperlinks, specified as '`on`' or '`off`'. Specify '`on`' to include code-to-model hyperlinks in the code generation report. The hyperlinks link code to the corresponding blocks, Stateflow® objects, and MATLAB functions in the model diagram. For more information see "Code-to-model" (Simulink Coder).

Example: `''IncludeHyperlinkInReport','on''`

Data Types: `char`

`GenerateTraceInfo` — Model-to-code highlighting

`'off'` | `'on'`

Model-to-code highlighting, specified as '`on`' or '`off`'. Specify '`on`' to include model-to-code highlighting in the code generation report. For more information see "Model-to-code" (Simulink Coder).

Example: `''GenerateTraceInfo','on''`

Data Types: `char`

`GenerateWebview` — Model Web view

`'off'` | `'on'`

Model Web view, specified as '`on`' or '`off`'. Specify '`on`' to include the model Web view in the code generation report. For more information, see "Generate model Web view" (Simulink Coder).

Example: `''GenerateWebview','on''`

Data Types: `char`

Traceability Report Contents

`GenerateTraceReport` — Summary of eliminated and virtual blocks
`'off'` | `'on'`

Summary of eliminated and virtual blocks, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of eliminated and virtual blocks in the code generation report. For more information, see "Eliminated / virtual blocks" (Simulink Coder).

Example: `''GenerateTraceReport','on''`

Data Types: `char`

`GenerateTraceReportSl` — Summary of Simulink blocks and the corresponding code location
`'off'` | `'on'`

Summary of the Simulink blocks and the corresponding code location, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of the Simulink blocks and the corresponding code location in the code generation report. For more information, see "Traceable Simulink blocks" (Simulink Coder).

Example: `''GenerateTraceReportSl','on''`

Data Types: `char`

`GenerateTraceReportsSf` — Summary of Stateflow objects and the corresponding code location
`'off'` | `'on'`

Summary of the Stateflow objects and the corresponding code location, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of Stateflow objects and the corresponding code location in the code generation report. For more information, see "Traceable Stateflow objects" (Simulink Coder).

Example: `''GenerateTraceReportSf','on''`

Data Types: `char`

`GenerateTraceReportEml` — Summary of MATLAB functions and the corresponding code location
`'off'` | `'on'`

Summary of the MATLAB functions and the corresponding code location, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of the MATLAB objects and the corresponding

code location in the code generation report. For more information, see “Traceable MATLAB functions” (Simulink Coder).

Example: `''GenerateTraceReportEml','on''`

Data Types: `char`

Metrics

`GenerateCodeMetricsReport` — Static code metrics
`'off'` | `'on'`

Static code metrics, specified as `'on'` or `'off'`. Specify `'on'` to include static code metrics in the code generation report. For more information, see “Static code metrics” (Simulink Coder).

Example: `''GenerateCodeMetricsReport','on''`

Data Types: `char`

See Also

`coder.report.close` | `coder.report.open`

Topics

“Reports for Code Generation” (Simulink Coder)
“Generate a Code Generation Report” (Simulink Coder)
“Generate Code Generation Report After Build Process” (Simulink Coder)

Introduced in R2012a

coder.report.open

Open existing HTML code generation report

Syntax

```
coder.report.open(model)
coder.report.open(subsystem)
```

Description

`coder.report.open(model)` opens a code generation report for the `model`. The build folder for the model must be present in the current working folder.

`coder.report.open(subsystem)` opens a code generation report for the `subsystem`. The build folder for the subsystem must be present in the current working folder.

Examples

Open code generation report for a model

After generating code for `rtwdemo_counter`, open a code generation report for the model.

```
coder.report.open('rtwdemo_counter')
```

Open code generation report for a subsystem

Open a code generation report for the subsystem ‘`Amplifier`’ in model ‘`rtwdemo_counter`’.

```
coder.report.open('rtwdemo_counter/Amplifier')
```

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: ‘`rtwdemo_counter`’

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: ‘`rtwdemo_counter/Amplifier`’

Data Types: `char`

See Also
`coder.report.close` | `coder.report.generate`

Topics
“Reports for Code Generation” (Simulink Coder)
“Open Code Generation Report” (Simulink Coder)

Introduced in R2012a

extmodeBackgroundRun

Perform external mode background activity

Syntax

```
errorCode = extmodeBackgroundRun();
```

Description

`errorCode = extmodeBackgroundRun();` performs external mode background activity, for example, retrieving packets from the network, running the packets protocol layer, and sending packets to the development computer.

Do not invoke the function in a thread with real-time constraints.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_BUSY` (-6) –– Resource busy detected, try later
- `EXTMODE_INV_MSG_FORMAT` (-7) –– Invalid message format detected by external mode communication protocol.
- `EXTMODE_INV_SIZE` (-8) –– Invalid size detected by the external mode communication protocol.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.
- `EXTMODE_NO_MEMORY` (-10) –– No memory available on the target hardware.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.
- `EXTMODE_PKT_CHECKSUM_ERROR` (-13) –– Checksum inconsistency detected by external mode communication protocol.
- `EXTMODE_PKT_RX_TIMEOUT_ERROR` (-14) –– Timeout error detected during the reception of a packet.
- `EXTMODE_PKT_TX_TIMEOUT_ERROR` (-15) –– Timeout error detected during the transmission of a packet.

See Also

`extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeEvent

External mode event trigger

Syntax

```
errorCode = extmodeEvent(eventId, simulationTime)
```

Description

`errorCode = extmodeEvent(eventId, simulationTime)` informs the external mode abstraction layer of the occurrence of an event.

`eventId` is the sample time ID of the model, for example, 0 for base rate, 1 for first subrate, and so on.

The function:

- Samples all signals associated with a given sample time.
- Stores signal values in a new packet buffer.
- Passes the packet buffer to the underlying transport layer for subsequent transmission to the development computer.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

For correct sampling of signal values, run the function immediately after *`model`*`_step()` for the corresponding sample time ID. You can invoke the function with different sample time IDs in separate threads because the function is thread-safe.

The `extmodeBackgroundRun` function performs the transmission of signal values to the development computer.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Input Arguments

`eventId` — Event ID
uint16_T

Sample time ID of the model, which is 0 for base rate, 1 for first subrate, 2 for second subrate, and so on.

`simulationTime` — Simulation time
real_T

Time when event occurs.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.
- `EXTMODE_NO_MEMORY` (-10) –– No memory available on the target hardware.

See Also

`extmodeBackgroundRun` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` |

`extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeGetFinalSimulationTime

Get final simulation time for external mode platform abstraction layer

Syntax

```
errorCode = extmodeGetFinalSimulationTime(finalTime);
```

Description

`errorCode = extmodeGetFinalSimulationTime(finalTime);` gets the model's final simulation time for the external mode platform abstraction layer. The function is a complementary function for `extmodeSetFinalSimulationTime`.

Output Arguments

`finalTime` — Final simulation time
real_T pointer

Final simulation time of model.

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` |

`extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeInit

Initialize external mode target connectivity

Syntax

```
errorCode = extmodeInit(extmodeInfo, finalTime);
```

Description

`errorCode = extmodeInit(extmodeInfo, finalTime);` initializes the external mode target connectivity, including the underlying communication stack.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Input Arguments

`extmodeInfo` — External mode information structure
RTWExtModeInfo structure

Model structure that contains information for the external mode simulation. `RTWExtModeInfo` is defined in *`matlabroot`*`/simulink/include/rtw_extmode.h`.

`finalTime` — Final simulation time
real_T pointer

If the model's final simulation time in the external mode abstraction layer is initialized, then `finalTime` is an output and the pointer location is updated with the initialized value. You might initialize the final simulation time through the `'-tf '` option detected by `extmodeParseArgs()` or `extmodeSetFinalSimulationTime()`

If the model's final simulation time in the external mode abstraction layer is not initialized, then `finalTime` is an input and the model's final simulation time in external mode is updated accordingly.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

"External Mode Simulation with XCP Communication" (Simulink Coder)
"Customize XCP Slave Software" (Simulink Coder)

Introduced in R2018a

extmodeParseArgs

Extract values of configuration parameters supported by external mode abstraction layer

Syntax

```
errorCode = extmodeParseArgs(argCount, argValues);
```

Description

`errorCode = extmodeParseArgs(argCount, argValues);` extracts the values of the configuration parameters that are supported by the external mode abstraction layer. The function parses the array of strings passed as input arguments. The array of strings is from the command-line arguments of the executable file running on the target hardware.

The external mode abstraction layer interprets only two options and passes the other arguments to `rtIOStreamOpen` for the initialization of the communication driver.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

If your target hardware does not support the parsing of command-line arguments, define the preprocessor macro `EXTMODE_DISABLE_ARGS_PROCESSING`. See information about parsing command-line arguments in “Other Platform Abstraction Layer Functionality” (Simulink Coder).

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Input Arguments

`argCount` — Number of arguments
int_T scalar

Number of elements in `argValues` array.

`argValues` — Command-line arguments
array of null-terminated strings

Command-line arguments of the executable file running on the target hardware. The external mode abstraction layer interprets only these options:

- `'-w'` – Enables the `extmodeWaitForStartRequest()` function, which waits for a model start request from Simulink in external mode. If you do not specify this option, the `extmodeWaitForStartRequest()` function has no effect.
- `'-tf` *`finalSimulationTime`*`'` – *`finalSimulationTime`* overrides the Simulink configuration parameter, `StopTime`.

If the command contains more options, they are passed to `rtIOStreamOpen` as configuration parameters for the communication driver.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeReset

Reset external mode target connectivity

Syntax

```
errorCode = extmodeReset();
```

Description

`errorCode = extmodeReset();` restores the external mode abstraction layer, including the communication stack, to the initial, default state.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeSetFinalSimulationTime

Set final simulation time in external mode platform abstraction layer

Syntax

```
errorCode = extmodeSetFinalSimulationTime(finalTime);
```

Description

`errorCode = extmodeSetFinalSimulationTime(finalTime);` sets the final simulation time of the model in the external mode platform abstraction layer.

In the `main` function of your external mode target application, before `extmodeInit`, you can call `extmodeSetFinalSimulationTime` to set the final simulation time if:

- You do not want to use `extmodeParseArgs`.
- Your target hardware does not support parsing of command-line arguments but you want to override `StopTime` from the target application.

`extmodeGetFinalSimulationTime` and `extmodeSetFinalSimulationTime` are complementary functions.

Input Arguments

`finalTime` — Final simulation time
real_T

Final simulation time of model.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeSimulationComplete

Check that external mode simulation is complete

Syntax

```
simComplete = extmodeSimulationComplete();
```

Description

`simComplete = extmodeSimulationComplete();` during an external mode simulation, checks whether the model simulation time has reached the final simulation time specified by the command-line `'-tf'` option or the Simulink configuration parameter, `StopTime`.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Output Arguments

`simComplete` — Simulation complete
true | false

`true` if model simulation time has reached the specified final simulation time. Otherwise, returns `false`.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeStopRequested

Check whether request to stop external mode simulation is received from model

Syntax

```
stopRequest = extmodeStopRequested();
```

Description

`stopRequest = extmodeStopRequested();` checks whether a request to stop the external mode simulation is received from the Simulink model on the development computer.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Output Arguments

`stopRequest` — Stop request
true | false

`true` if request to stop external mode simulation is received. Otherwise, returns `false`.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

extmodeWaitForHostRequest

Wait for request from development computer to start or stop external mode simulation

Syntax

```
errorCode = extmodeWaitForHostRequest(timeoutInMicroseconds);
```

Description

`errorCode = extmodeWaitForHostRequest(timeoutInMicroseconds);` waits for a start or stop request from the development computer and times out when the timeout value is reached.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation. Use the function during initialization because the function is a blocking function.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer” (Simulink Coder).

Input Arguments

`timeoutInMicroseconds` — Timeout
uint32_T

Specifies the timeout value. If the value is set to `EXTMODE_WAIT_FOREVER`, the function waits indefinitely. If `'-w'` is not extracted by `extmodeParseArgs()`, the function has no effect.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_TIMEOUT_ERROR` (-100) –– External mode timeout error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested`

Topics

“External Mode Simulation with XCP Communication” (Simulink Coder)
“Customize XCP Slave Software” (Simulink Coder)

Introduced in R2018a

findBuildArg

Find a specific build argument in model build information

Syntax

```
[identifier,value] = findBuildArg(buildinfo,buildArgName)
```

Description

`[identifier,value] = findBuildArg(buildinfo,buildArgName)` searches for a build argument from the build information.

If the build argument is present in the model build information, the function returns the name and value.

Examples

Find Build Argument in Build Information

Find a build argument and its value stored in build information `myModelBuildInfo`. Then, view the argument identifier and value.

```
load buildInfo.mat
myModelBuildInfo = buildInfo;
myBuildArgExtmodeStaticAlloc = 'EXTMODE_STATIC_ALLOC';
[buildArgId buildArgValue] = findBuildArg(buildInfo, ...
   myBuildArgExtmodeStaticAlloc);
```

```
>> buildArgId

buildArgId =

    'EXTMODE_STATIC_ALLOC'
```

```
>> buildArgValue

buildArgValue =

    '0'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`buildArgName` — Name of build argument to find in build information
character vector | string scalar

To get the build argument identifiers from the build information, use the `getBuildArgs` function.

Output Arguments

`identifier` — Name of the build argument
character vector | string scalar

`value` — Value of the build argument
character vector | string scalar

See Also

`getBuildArgs`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2014a

findIncludeFiles

Find and add include (header) files to model build information

Syntax

```
findIncludeFiles(buildinfo,extPatterns)
```

Description

`findIncludeFiles(buildinfo,extPatterns)` searches for and adds include files to the build information.

Use the `findIncludeFiles` function to:

- Search for include files in source and include paths from the build information.
- Apply the optional *`extPatterns`* argument to specify file name extension patterns for search.
- Add the found files with their full paths to the build information.
- Delete duplicate include file entries from the build information.

Examples

Find and Add Include Files to Build Information

Find include files with file name extension `.h` that are in the build information, `myModelBuildInfo`. Add the full paths for these files to the build information. View the include files from the build information.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,{fullfile(pwd,...
    'mycustomheaders')},'myheaders');
findIncludeFiles(myModelBuildInfo);
headerfiles = getIncludeFiles(myModelBuildInfo,true,false);
```

```
>> headerfiles

headerfiles =

    'W:\work\mycustomheaders\myheader.h'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`extPatterns` — Patterns of file name extensions that specify files for the search
`'*.h'` (default) | cell array of character vectors | string array

To specify files for the search, the character vectors or strings in the *`extPatterns`* argument:

- Must start with an asterisk immediately followed by a period (*.)
- Can include a combination of alphanumeric and underscore (_) characters

Example: `'*.h' '*.hpp' '*.x*'`

See Also

`addIncludeFiles` | `getIncludeFiles` | `packNGo`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006b

getBuildArgs

Get build arguments from model build information

Syntax

```
[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs,
excludeGroupIDs)
```

Description

`[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs, excludeGroupIDs)` returns build argument identifiers and values from model build information.

The function requires the *`buildinfo`*, *`identifiers`*, and *`values`* arguments. You can use optional *`includeGroupIDs`* and *`excludeGroupIDs`* arguments. These optional arguments let you include or exclude groups selectively from the build arguments returned by the function.

If you choose to specify *`excludeGroupIDs`* and omit *`includeGroupIDs`*, specify a null character vector (`''`) for *`includeGroupIDs`*.

Examples

Get Build Arguments from Build Information

After you build a model, the build information is available in the `buildInfo.mat` file. Retrieve the build arguments from the build information object.

```
load buildInfo.mat
[buildArgIds,buildArgValues] = getBuildArgs(buildInfo);
```

To get the value of a single build argument from the build information, you can use the `findBuildArg` function.

To view the build argument identifiers, enter:

```
buildArgIds
```

To view the build argument values, enter:

```
buildArgValues
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`includeGroupIDs` — Group identifiers of build arguments to include in the return from the function
cell array of character vectors | string

To use the *`includeGroupIDs`* argument, view available build argument identifier groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroupIDs` — Group identifiers of build arguments to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroupIDs`* argument, view available build argument identifier groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`identifiers` — Names of the build arguments
cell array of character vectors

`values` — Values of the build arguments
cell array of character vectors

See Also

`findBuildArg`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2014a

getCodeDescriptor

Create `coder.codedescriptor.CodeDescriptor` object for model

Syntax

```
getCodeDescriptor(model)
getCodeDescriptor(folder)
```

Description

`getCodeDescriptor(model)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified model.

`getCodeDescriptor(folder)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified build folder.

Examples

Create a Code Descriptor Object Using Model Name

Create a `coder.codedescriptor.CodeDescriptor` object by using model name:

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

Create a Code Descriptor Object Using Build Folder

Create a `coder.codedescriptor.CodeDescriptor` object by using build folder:

```
codeDescObj = coder.getCodeDescriptor('C:\Users\Desktop\work\rtwdemo_comments_ert_rtw')
```

Input Arguments

`model` — Name of the model

string

Model object or name for which to obtain the `coder.codedescriptor.CodeDescriptor` object. You can get the `coder.codedescriptor.CodeDescriptor` object only for the top model if the model has referenced models.

Example: `rtwdemo_comments`

Data Types: `string`

`folder` — Build folder of the model

string

Build folder of the model for which to obtain the `coder.codedescriptor.CodeDescriptor` object. You can get the `coder.codedescriptor.CodeDescriptor` object only for the top model if the model has referenced models.

Example: `C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw`

Data Types: `string`

See Also

`coder.codedescriptor.CodeDescriptor`

Topics

“Get Code Description of Generated Code” (Simulink Coder)

Introduced in R2018a

getCompileFlags

Get compiler options from model build information

Syntax

```
options = getCompileFlags(buildinfo,includeGroups,excludeGroups)
```

Description

`options = getCompileFlags(buildinfo,includeGroups,excludeGroups)` returns compiler options from the build information.

The function requires the *`buildinfo`* argument. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the compiler options returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Compiler Options from Build Information

Get the compiler options stored in the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
   'OPTS');
compflags = getCompileFlags(myModelBuildInfo);
```

```
>> compflags

compflags =
```

```
    '-Zi -Wall'    '-O3'
```

Get Compiler Options with Include Group Argument

Get the compiler options stored with the group name `Debug` in the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
   {'Debug' 'MemOpt'});
compflags = getCompileFlags(myModelBuildInfo,'Debug');
```

```
>> compflags

compflags =

    '-Zi -Wall'
```

Get Compiler Options with Exclude Group Argument

Get the compiler options stored in the build information `myModelBuildInfo`, except those options with the group name `Debug`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
   {'Debug' 'MemOpt'});
compflags = getCompileFlags(myModelBuildInfo,'','Debug');
```

```
>> compflags

compflags =
```

```
'-O3'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo` object

`includeGroups` — Group names of compiler options to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of compiler options to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`options` — Compiler options from the build information
cell array of character vectors

See Also

`addCompileFlags` | `getDefines` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

getDefines

Get preprocessor macro definitions from model build information

Syntax

```
[macrodefs,identifiers,values] = getDefines(buildinfo,includeGroups,
excludeGroups)
```

Description

`[macrodefs,identifiers,values] = getDefines(buildinfo,includeGroups, excludeGroups)` returns preprocessor macro definitions from the build information.

The function requires the *`buildinfo`*, *`macrodefs`*, *`identifiers`*, and *`values`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the preprocessor macro definitions returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Macro Definitions from Build Information

Get the preprocessor macro definitions stored in the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'},'OPTS');
[defs,names,values] = getDefines(myModelBuildInfo);
```

```
>> defs
```

```
defs =

    '-DPROTO=first'    '-DDEBUG'    '-Dtest'    '-DPRODUCTION'

>> names

names =

    'PROTO'
    'DEBUG'
    'test'
    'PRODUCTION'

>> values

values =

    'first'
    ''
    ''
    ''
```

Get Macro Definitions with Include Group Argument

Get the preprocessor macro definitions stored with the group name `Debug` in the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
   {'Debug' 'Debug' 'Debug' 'Release'});
[defs,names,values] = getDefines(myModelBuildInfo,'Debug');
```

```
>> defs

defs =

    '-DPROTO=first'    '-DDEBUG'     '-Dtest'
```

Get Macro Definitions with Exclude Group Argument

Get the preprocessor macro definitions stored in the build information `myModelBuildInfo`, except those definitions with the group name `Debug`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
   {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
   {'Debug' 'Debug' 'Debug' 'Release'});
[defs,names,values] = getDefines(myModelBuildInfo,'','Debug');
```

```
>> defs

defs =

    '-DPRODUCTION'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`includeGroups` — Group names of macro definitions to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of macro definitions to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`macrodefs` — Macro definitions from the build information
cell array of character vectors

The *`macrodefs`* provide the complete macro definitions with a `-D` prefix. When the function returns a definition:

- If the `-D` was not specified when the definition was added to the build information, prepends a `-D` to the definition.
- Changes a lowercase `-d` to `-D`.

`identifiers` — Names of the macros from the build information
cell array of character vectors

`values` — Values assigned to the macros from the build information
cell array of character vectors

The *`values`* provide anything specified to the right of the first equal sign in the macro definition. The default is an empty character vector (`''`).

See Also

`addDefines` | `getCompileFlags` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

getFullFileList

Get list of files from model build information

Syntax

```
[fPathNames,names] = getFullFileList(buildinfo,fcase)
```

Description

`[fPathNames,names] = getFullFileList(buildinfo,fcase)` returns the fully qualified paths and names of files, or files of a selected type (source, include, or nonbuild), from the build information.

The function requires the *`buildinfo`*, *`fPathNames`*, and *`names`* arguments. You can use the optional *`fcase`* argument. This optional argument lets you include or exclude file cases selectively from file list returned by the function.

To ensure that header files are added to the file list (for example, header files in the _sharedutils (Simulink Coder) folder), run `findIncludeFiles` before `getFullFileList`.

The `packNGo` function calls `getFullFileList` to return a list of files in the build information before processing files for packaging.

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. The `getFullFileList` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

Examples

Get Full File List of Files

After building a model and loading the generated `buildInfo.mat` file, you can list the files stored in a build information variable, `myModelBuildInfo`. This example returns information for the current model and descendants (submodels).

```
myModelBuildInfo = RTW.BuildInfo;
findIncludeFiles(myModelBuildInfo);
[fPathNames,names] = getFullFileList(myModelBuildInfo);
```

Get Full File List of Source Files

If you use an *`fcase`* option, you limit the listing to the files stored in the `myModelBuildInfo` variable for the current model. This example returns information for the current model only (not for descendants or submodels).

```
[fPathNames,names] = getFullFileList(myModelBuildInfo,'source');
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`fcase` — File case to return from the build information
`''` (default) | `'source'` | `'include'` | `'nonbuild'`

The *`fcase`* argument selects whether the function returns the full list for files in the build information or returns selected cases of files. If you omit the argument or specify a null character vector (`''`), the function returns the files from the build information.

Specify	Function Action
`'source'`	Returns source files from the build information.
`'include'`	Returns include files from the build information.
`'nonbuild'`	Returns nonbuild files from the build information.

Example: `'source'`

Output Arguments

`fPathNames` — Fully qualified file paths from the build information
cell array of character vectors

`names` — File names from the build information
cell array of character vectors

See Also

`findIncludeFiles`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2008a

getIncludeFiles

Get include files from model build information

Syntax

```
files = getIncludeFiles(buildinfo,concatenatePaths,
replaceMatlabroot,includeGroups,excludeGroups)
```

Description

`files = getIncludeFiles(buildinfo,concatenatePaths, replaceMatlabroot,includeGroups,excludeGroups)` returns the names of include files from the build information.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the include files returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getIncludeFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getIncludeFiles`.

Examples

Get Include Paths and Files from Build Information

Get the include paths and file names from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo,{'etc.h' 'etc_private.h' ...
   'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...
   '/common/lib'},{'etc' 'etc' 'shared'});
incfiles=getIncludeFiles(myModelBuildInfo,true,false);
```

```
>> incfiles

incfiles =

    [1x22 char]    [1x36 char]    [1x21 char]
```

Get Include Paths and Files with Include Group Argument

Get the names of include files in group `etc` from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo,{'etc.h' 'etc_private.h' ...
   'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...
   '/common/lib'},{'etc' 'etc' 'shared'});
incfiles = getIncludeFiles(myModelBuildInfo,false,false, ...
   'etc');
```

```
>> incfiles

incfiles =

    'etc.h'     'etc_private.h'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of include paths and files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of include paths and files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: ' '

Output Arguments

`files` — Names of include files from the build information
cell array of character vectors

The names of include files that you add with the `addIncludeFiles` function. If you call the `packNGo` function, the names include files that `packNGo` found and added while packaging model code.

See Also

`addIncludeFiles` | `findIncludeFiles` | `getIncludePaths` | `getSourceFiles` | `getSourcePaths` | `updateFilePathsAndExtensions`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

getIncludePaths

Get include paths from model build information

Syntax

```
paths = getIncludePaths(buildinfo,replaceMatlabroot,includeGroups,
excludeGroups)
```

Description

`paths = getIncludePaths(buildinfo,replaceMatlabroot,includeGroups, excludeGroups)` returns the names of include file paths from the build information.

The function requires the *`buildinfo`* and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the include paths returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Include Paths from Build Information

Get the include paths from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,{'/etc/proj/etclib' ...
   '/etcproj/etc/etc_build' '/common/lib'}, ...
   {'etc' 'etc' 'shared'});
incpaths = getIncludePaths(myModelBuildInfo,false);


>> incpaths
```

```
incpaths =

    '\etc\proj\etclib'   [1x22 char]    '\common\lib'
```

Get Include Paths with Include Group Argument

Get the paths in group `shared` from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,{'/etc/proj/etclib' ...
   '/etcproj/etc/etc_build' '/common/lib'}, ...
   {'etc' 'etc' 'shared'});
incpaths = getIncludePaths(myModelBuildInfo,false,'shared');
```

```
>> incpaths

incpaths =

    '\common\lib'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return from the function
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output that it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of include paths to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of include paths to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`paths` — Paths of include files from the build information
cell array of character vectors

See Also

`addIncludePaths` | `getIncludeFiles` | `getSourceFiles` | `getSourcePaths`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

getLinkFlags

Get link options from model build information

Syntax

```
options = getLinkFlags(buildinfo,includeGroups,excludeGroups)
```

Description

`options = getLinkFlags(buildinfo,includeGroups,excludeGroups)` returns linker options from the build information.

The function requires the *`buildinfo`* argument. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the compiler options returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Linker Options from Build Information

Get the linker options from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,{'-MD -Gy' '-T'},'OPTS');
linkflags = getLinkFlags(myModelBuildInfo);
```

```
>> linkflags

linkflags =
```

```
    '-MD -Gy'    '-T'
```

Get Linker Options with Include Group Argument

Get the linker options with the group name `Debug` from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,{'-MD -Gy' '-T'}, ...
   {'Debug' 'MemOpt'});
linkflags = getLinkFlags(myModelBuildInfo,{'Debug'});
```

```
>> linkflags

linkflags =

    '-MD -Gy'
```

Get Linker Options with Exclude Group Argument

Get the linker options from the build information `myModelBuildInfo`, except those options with the group name `Debug`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo,{'-MD -Gy' '-T'}, ...
   {'Debug' 'MemOpt'});
linkflags = getLinkFlags(myModelBuildInfo,'',{'Debug'});
```

```
>> linkflags

linkflags =
```

```
'-T'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo` object

`includeGroups` — Group names of linker options to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of linker options to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`options` — Linker options from the build information
cell array of character vectors

See Also

`addLinkFlags` | `getCompileFlags` | `getDefines`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

getNonBuildFiles

Get nonbuild-related files from model build information

Syntax

```
files = getNonBuildFiles(buildinfo, concatenatePaths,
replaceMatlabroot,includeGroups,excludeGroups)
```

Description

`files = getNonBuildFiles(buildinfo, concatenatePaths, replaceMatlabroot,includeGroups,excludeGroups)` returns the names of non-build files from the build information, such as DLL files required for a final executable or a README file.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the non-build files returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getNonBuildFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getIncludeFiles`.

Examples

Get Nonbuild Files from Build Information

Get the nonbuild file names stored in the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo,{'readme.txt' 'myutility1.dll' ...
   'myutility2.dll'});
nonbuildfiles = getNonBuildFiles(myModelBuildInfo,false,false);
```

```
>> nonbuildfiles

nonbuildfiles =

    'readme.txt'    'myutility1.dll'    'myutility2.dll'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return from function
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return from function
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output that it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of non-build files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of non-build files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`files` — Names of non-build files from the build information
cell array of character vectors

See Also

`addNonBuildFiles`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2008a

getSourceFiles

Get source files from model build information

Syntax

```
srcfiles = getSourceFiles(buildinfo,concatenatePaths,
replaceMatlabroot,includeGroups,excludeGroups)
```

Description

`srcfiles = getSourceFiles(buildinfo,concatenatePaths, replaceMatlabroot,includeGroups,excludeGroups)` returns the names of source files from the build information.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the source files returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getSourceFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getSourceFiles`.

Examples

Get Source Files from Build Information

Get the source paths and file names from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
   {'test1.c' 'test2.c' 'driver.c'},'', ...
   {'Tests' 'Tests' 'Drivers'});
srcfiles = getSourceFiles(myModelBuildInfo,false,false);
```

```
>> srcfiles

srcfiles =

    'test1.c'    'test2.c'    'driver.c'
```

Get Source Files with Include Group Argument

Get the names of source files in group `tests` from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo,{'test1.c' 'test2.c'...
   'driver.c'}, {'/proj/test1' '/proj/test2'...
   '/drivers/src'}, {'tests', 'tests', 'drivers'});
incfiles = getSourceFiles(myModelBuildInfo,false,false,...
   'tests');
```

```
>> incfiles

incfiles =

    'test1.c'    'test2.c'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of source files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of source files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`srcfiles` — Names of source files from the build information
cell array of character vectors

The names of source files that you add with the `addSourceFiles` function. If you call the `packNGo` function, the names include files that `packNGo` found and added while packaging model code.

See Also

`addSourceFiles` | `getIncludeFiles` | `getIncludePaths` | `getSourcePaths` | `updateFilePathsAndExtensions`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

getSourcePaths

Get source paths from model build information

Syntax

```
srcpaths = getSourcePaths(buildinfo,replaceMatlabroot,includeGroups,
excludeGroups)
```

Description

`srcpaths = getSourcePaths(buildinfo,replaceMatlabroot,includeGroups, excludeGroups)` returns the names of source file paths from the build information.

The function requires the *`buildinfo`* and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the source paths returned by the function.

If you choose to specify *`excludeGroups`* and omit *`includeGroups`*, specify a null character vector (`''`) for *`includeGroups`*.

Examples

Get Source Paths from Build Information

Get the source paths from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,{'/proj/test1' ...
   '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...
   'drivers'});
srcpaths = getSourcePaths(myModelBuildInfo,false);


>> srcpaths
```

```
srcpaths =

   '\proj\test1'    '\proj\test2'      '\drivers\src'
```

Get Source Paths with Include Group Argument

Get the paths in group `tests` from the build information, `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,{'/proj/test1' ...
   '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...
   'drivers'});
srcpaths = getSourcePaths(myModelBuildInfo,true,'tests');


>> srcpaths

srcpaths =

     '\proj\test1'    '\proj\test2'
```

Get Source Paths from Build Information

Get a source path from the build information, `myModelBuildInfo`. First, get the path without replacing `$(MATLAB_ROOT)` with an absolute path. Then, get it with replacement. Here, the MATLAB root folder is `\\myserver\myworkspace\matlab`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, fullfile(matlabroot, ...
   'rtw', 'c', 'src'));
srcpaths = getSourcePaths(myModelBuildInfo,false);


>> srcpaths{:}

ans =

$(MATLAB_ROOT)\rtw\c\src

>> srcpaths = getSourcePaths(myModelBuildInfo,true);
```

```
>> srcpaths{:}

ans =

\\myserver\myworkspace\matlab\rtw\c\src
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo` object

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of source paths to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups with `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of source paths to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups with `myGroups = getGroups(buildInfo)`.

Example: ' '

Output Arguments

`srcpaths` — Paths of source files from the build information
cell array of character vectors

See Also

`addSourcePaths` | `getIncludeFiles` | `getIncludePaths` | `getSourceFiles`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

model_initialize

Initialization entry-point function in generated code for Simulink model

Syntax

```
void model_initialize(void)
```

Calling Interfaces

The calling interface generated for this function differs depending on the value of the model parameter **Code interface packaging** (Simulink Coder):

- `C++ class` (default for C++ language) — Generated function is encapsulated into a C ++ class method. Required model data is encapsulated into C++ class attributes.
- `Nonreusable function` (default for C language) — Generated function passes `(void)`. Model data structures are statically allocated, global, and accessed directly in the model code.
- `Reusable function` — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the *`model`*`.h` header file.

  For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the function. They can be included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.

  For a GRT-based model, the generated *`model`*`.c` source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, you can use the **Use dynamic memory allocation for model initialization** parameter to control whether an allocation function is generated.

  - When set, you can restart code generated from the model from a single execution instance. The sequence of function calls from the `main.c` is `allocfcn`, `model_init`, `model_term`, `allocfcn`, `model_init`, `model_term`.

- When cleared,

Note If you have an Embedded Coder license, for `Nonreusable function` code interface packaging, you can use the Code Mapping Editor to customize the name of the initialize function interface. See “Override Default Naming for Individual C Entry-Point Functions”.

Description

The generated *`model`*`_initialize` function contains initialization code for a Simulink model and should be called once at the start of your application code.

Do not use the *`model`*`_initialize` function to reset the real-time model data structure (rtM).

See Also

`model_step` | `model_terminate`

Topics

“Configure Code Generation for Model Entry-Point Functions” (Simulink Coder)
“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)

Introduced before R2006a

model_step

Step routine entry point in generated code for Simulink model

Syntax

`void *model*_step(void)`

`void *model*_step*N*(void)`

Calling Interfaces

The `*model*_step` default function prototype varies depending on the **Treat each discrete rate as a separate task** (Simulink) (`EnableMultiTasking`) parameter specified for the model:

Parameter Value	Function Prototype
`Off` (single rate or multirate)	`void *model*_step(void);`
`On` (multirate)	`void *model*_step*N* (void);` (*N* is a task identifier)

The calling interface generated for this function also differs depending on the value of the model parameter **Code interface packaging** (Simulink Coder):

- `C++ class` (default for C++ language) — Generated function is encapsulated into a C ++ class method. Required model data is encapsulated into C++ class attributes.
- `Nonreusable function` (default for C language) — Generated function passes `(void)`. Model data structures are statically allocated, global, and accessed directly in the model code.
- `Reusable function` — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the `*model*.h` header file.

  For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the function. They can be

included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.

Note If you have an Embedded Coder license:

- For `Nonreusable function` code interface packaging, you can use the Configure C Step Function Interface dialog box to customize a C step function interface. See "Override Default C Step Function Interface""Customize Generated C Function Interfaces".
- For `C++ class` code interface packaging, you can use the **Configure C++ Class Interface** button and related controls on the **Interface** pane of the Configuration Parameters dialog box. For more information, see "Customize Generated C++ Class Interfaces".

Description

The generated *`model`*`_step` function contains the output and update code for the blocks in a Simulink model. The *`model`*`_step` function computes the current value of the 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.

Under 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 one or more of these conditions are true, the program runs indefinitely.

For a GRT or ERT-based model, the software generates a *`model`*`_step` function when the **Single output/update function** configuration option is selected (the default) in the Configuration Parameters dialog box.

`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. For a description of how calls to

*model*_step are generated and scheduled, see “rt_OneStep and Scheduling Considerations”.

Note If the **Single output/update function** configuration option is not selected, the software generates the following model entry point functions in place of *model*_step:

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

---

See Also

model_initialize | model_terminate

Topics

“Configure Code Generation for Model Entry-Point Functions” (Simulink Coder)

Introduced before R2006a

model_terminate

Termination entry point in generated code for Simulink model

Syntax

```
void model_terminate(void)
```

Calling Interfaces

The calling interface generated for this function also differs depending on the value of the model parameter **Code interface packaging** (Simulink Coder):

- `C++ class` (default for C++ language) — Generated function is encapsulated into a C ++ class method. Required model data is encapsulated into C++ class attributes.
- `Nonreusable function` (default for C language) — Generated function passes `(void)`. Model data structures are statically allocated, global, and accessed directly in the model code.
- `Reusable function` — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the *`model`*`.h` header file.

  For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the function. They can be included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.

Description

The generated *`model`*`_terminate` function contains the termination code for a Simulink model 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.

For an ERT-based model, the code generator produces the *model*_terminate function for a model when the **Terminate function required** configuration option is selected (the default) in the Configuration Parameters dialog box. 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_step

Topics

“Configure Code Generation for Model Entry-Point Functions” (Simulink Coder)
“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)

Introduced before R2006a

packNGo

Package generated code in zip file for relocation

Syntax

```
packNGo(buildInfo,{Name,Value})
```

Description

`packNGo(buildInfo,{Name,Value})` packages the code files in a compressed zip file so that you can relocate, unpack, and rebuild them in another development environment. The list of name-value pairs is optional.

The types of code files in the zip file include:

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)
- MAT-file that contains the build information object (`.mat` file)
- Nonbuild-related files (for example, `.dll` files and `.txt` informational files) required for a final executable
- Build-generated binary files (for example, executable `.exe` file or dynamic link library `.dll`).

  The code generator includes the build-generated binary files (if present) in the zip file. The **ignoreFileMissing** property does not apply to build-generated binary files.

Use this function to relocate files. You can then recompile the files for a specific target environment or rebuild them in a development environment in which MATLAB is not installed. By default, the function packages the files as a flat folder structure in a zip file within the code generation folder. You can customize the output by specifying name-value pairs. After relocating the zip file, use a standard zip utility to unpack the compressed file.

The `packNGo` function can potentially modify the build information passed in the first `packNGo` argument. As part of code packaging, `packNGo` can find additional files from

source and include paths recorded in the build information. When these files are found, `packNGo` adds them to the build information.

Examples

Run packNGo from Command Window

After the build process is complete, you can run packNGo from the Command Window. Use `packNGo` for zip file packaging of generated code in the file `portzingbit.zip`. Maintain the relative file hierarchy.

1. Change folders to the code generation folder. For example, using MATLAB Coder, `codegen/dll/zingbit`, or for Simulink code generation, `zingbit_grt_rtw`.
2. Load the `buildInfo` object that describes the build.
3. Run `packNGo` with property settings for `packType` and `fileName`.

```
cd codegen/dll/zingbit;
load buildInfo.mat
packNGo(buildInfo,{'packType', 'hierarchical', ...
   'fileName','portzingbit'});
```

Configure packNGo in the Simulink Editor

If you configure zip file packaging from the code generation pane, the code generator uses `packNGo` to output a zip file during the build process.

1. Select **Code Generation** > **Package code and artifacts**. Optionally, provide a **Zip file name**. To apply the changes, click **OK**.
2. Build the model. At the end of the build process, the code generator outputs the zip file. The folder structure in the zip file is hierarchical.

Configure packNGo for Simulink from the Command Line

If you configure zip file packaging with `set_param`, the code generator uses `packNGo` to output a zip file during the build process.

Use `set_param` to configure zip file packaging for model `zingbit` in the file `zingbit.zip`.

```
set_param('zingbit','PostCodeGenCommand', ...
   'packNGo(buildInfo);');
```

Input Arguments

`buildInfo` — Object that provides build information
`buildInfo` object

During the build process, the code generator places `buildInfo.mat` in the code generation folder. This MAT-file contains the `buildInfo` object. The object provides information that `packNGo` uses to produce the zip file.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `{'packType','flat','nestedZipFiles',true}`

`packType` — Determines whether the primary zip file contains secondary zip files or folders
`'flat'` (default) | `'hierarchical'`

If `'flat'`, package the generated code files in a zip file as a single, flat folder.

If `'hierarchical'`, package the generated code files hierarchically in a primary zip file.

Example: `{'packType','flat'}`

`nestedZipFiles` — Determines whether the paths for files in the secondary zip files are relative to the root folder of the primary zip file
`true` (default) | false

If `true`, create a primary zip file that contains three secondary zip files:

- `mlrFiles.zip` — Files in your *`matlabroot`* folder tree

- `sDirFiles.zip` — Files in and under your code generation folder
- `otherFiles.zip` — Required files not in the *`matlabroot`* or `start` folder trees

If `false`, create a primary zip file that contains folders, for example, your code generation folder and *`matlabroot`*.

Example: `{'nestedZipFiles',true}`

`fileName` — Specifies a file name for the primary zip file
`'modelOrFunctionName``.zip'` (default) | *`'myName'`*

If you do not specify the `'fileName'`-value pair, the function packages the files in a zip file named *`modelOrFunctionName`*`.zip` and places the zip file in the code generation folder.

If you specify `'fileName'` with the value, *`'myName'`*, the function creates *`myName`*`.zip` in the code generation folder.

To specify another location for the primary zip file, provide the absolute path to the location, *`fullPath/myName`*`.zip`

Example: `{'fileName','/home/user/myModel.zip'}`

`minimalHeaders` — Selects whether to include only the minimal header files
`true` (default) | `false`

If `true`, include only the minimal header files required to build the code in the zip file.

If `false`, include header files found on the include path in the zip file.

Example: `{'minimalHeaders',true}`

`includeReport` — Selects whether to include the `html` folder for your code generation report
`false` (default) | `true`

If `false`, do not include the `html` folder in the zip file.

If `true`, include the `html` folder in the zip file.

Example: `{'includeReport',false}`

`ignoreParseError` — Instruct packNGo not to terminate on parse errors
false (default) | true

If `false`, terminate on parse errors.

If `true`, do not terminate on parse errors.

Example: `{'ignoreParseError',false}`

`ignoreFileMissing` — Instruct packNGo not to terminate if files are missing
false (default) | true

If `false`, terminate on missing file errors.

If `true`, do not terminate on missing files errors.

Example: `{'ignoreFileMissing',false}`

Limitations

- The function operates on source files only, such as `*.c`, `*.cpp`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
- Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if those files are not used.

See Also

Introduced in R2006b

rsimgetrtp

Global model parameter structure

Syntax

```
parameter_structure = rsimgetrtp('model')
```

Description

`parameter_structure = rsimgetrtp('model')` forces a block update diagram action for *`model`*, a model for which you are running rapid simulations, and returns the global parameter structure for that model. The function includes tunable parameter information in the parameter structure.

The model parameter structure contains the following fields:

Field	Description
`modelChecksum`	A four-element vector that encodes the structure. The code generator uses the *`checksum`* to check whether the structure has changed since the RSim executable was generated. If you delete or add a block, and then generate a new version of the structure, the new *`checksum`* will not match the original *`checksum`*. The RSim executable detects this incompatibility in model parameter structures and exits to avoid returning incorrect simulation results. If the structure changes, you must regenerate code for the model.
`parameters`	A structure that defines model global parameters.

The `parameters` substructure includes the following fields:

Field	Description
`dataTypeName`	Name of the parameter data type, for example, `double`

Field	Description
`dataTypeID`	An internal data type identifier
`complex`	Value 1 if parameter values are complex and 0 if real
`dtTransIdx`	Internal use only
`values`	Vector of parameter values
`structParamInfo`	Information about structure and bus parameters in the model

The `structParamInfo` substructure contains these fields:

Field	Description
`Identifier`	Name of the parameter
`ModelParam`	Value 1 if parameter is a model parameter and 0 if it is a block parameter
`BlockPath`	Block path for a block parameter. This field is empty for model parameters.
`CAPIIdx`	Internal use only

It is recommended that you do not modify fields in `structParamInfo`.

The function also includes an array of substructures `map` that represents tunable parameter information with these fields:

Field	Description
`Identifier`	Parameter name
`ValueIndicies`	Vector of indices to parameter values
`Dimensions`	Vector indicating parameter dimensions

Examples

Return global parameter structure for model `rtwdemo_rsimtf` to `param_struct`:

```
rtwdemo_rsimtf
param_struct = rsimgetrtp('rtwdemo_rsimtf')

param_struct =
```

```
    modelChecksum: [1.7165e+009 3.0726e+009 2.6061e+009
2.3064e+009]
       parameters: [1x1 struct]
```

See Also

`rsimsetrtpparam`

Topics

“Create a MAT-File That Includes a Model Parameter Structure” (Simulink Coder)
“Update Diagram and Run Simulation” (Simulink)
“Default parameter behavior” (Simulink Coder)
“Block Authoring and Simulation Integration” (Simulink)
“Tune Parameters” (Simulink Coder)

Introduced in R2006a

rsimsetrtpparam

Set parameters of `rtP` model parameter structure

Syntax

```
rtP = rsimsetrtpparam(rtP,idx)
rtP = rsimsetrtpparam(rtP,'paramName',paramValue)
rtP = rsimsetrtpparam(rtP,idx,'paramName',paramValue)
```

Description

`rtP = rsimsetrtpparam(rtP,idx)` expands the `rtP` structure to have `idx` sets of parameters. The `rsimsetrtpparam` utility defines the values of an existing `rtP` parameter structure. The `rtP` structure matches the format of the structure returned by `rsimgetrtp('`*`modelName`*`')`.

`rtP = rsimsetrtpparam(rtP,'paramName',paramValue)` takes an `rtP` structure with tunable parameter information and sets the values associated with `'paramName'` to be `paramValue` if possible. There can be more than one name-value pair.

`rtP = rsimsetrtpparam(rtP,idx,'paramName',paramValue)` takes an `rtP` structure with tunable parameter information and sets the values associated with `'paramName'` to be `paramValue` in the nth `idx` parameter set. There can be more than one name-value pair. If the `rtP` structure does not have `idx` parameter sets, the first set is copied and appended until there are `idx` parameter sets. Subsequently, the nth `idx`set is changed.

Examples

Expand Parameter Sets

Expand the number of parameter sets in the `rtp` structure to 10.

```
rtp = rsimsetrtpparam(rtp,10);
```

Add Parameter Sets

Add three parameter sets to the parameter structure `rtp`.

```
rtp = rsimsetrtpparam(rtp,idx,'X1',iX1,'X2',iX2,'Num',iNum);
```

Input Arguments

`rtP` — A parameter structure that contains the sets of parameter names and their respective values
parameter structure

`idx` — An index used to indicate the number of parameter sets in the `rtP` structure
index of parameter sets

`paramValue` — The value of the `rtP` parameter `paramName`
value of `paramName`

`paramName` — The name of the parameter set to add to the `rtP` structure
name of the parameter set

Output Arguments

`rtP` — An expanded `rtP` parameter structure that contains `idx` additional parameter sets defined by the `rsimsetrtpparam` function call
expanded `rtP` parameter structure

See Also

`rsimgetrtp`

Topics

“Create a MAT-File That Includes a Model Parameter Structure” (Simulink Coder)
“Update Diagram and Run Simulation” (Simulink)
“Default parameter behavior” (Simulink Coder)
“Block Authoring and Simulation Integration” (Simulink)
“Tune Parameters” (Simulink Coder)

Introduced in R2009b

rtw_precompile_libs

Rebuild precompiled libraries within model without building model

Syntax

`rtw_precompile_libs(model,build_spec)`

Description

`rtw_precompile_libs(model,build_spec)` builds libraries within *`model`*, according to the *`build_spec`* field values, and places the libraries in a precompiled folder. Model builds that use the template makefile approach support the `rtw_precompile_libs` function. Toolchain approach model builds do not support the `rtw_precompile_libs` function.

Examples

Precompile Libraries for Model

Build the libraries in *`my_model`* without building *`my_model`*.

```
% Specify the library suffix
if isunix
   suffix = '_std.a';
elseif ismac
   suffix = '_std.a';
else
   suffix = '_vcx64.lib';
end
open_system(my_model);
set_param(my_model, 'TargetLibSuffix',suffix);

% Set the precompiled library folder
set_param(my_model, 'TargetPreCompLibLocation',fullfile(pwd,'lib'));
```

```
% Define a build specification that specifies
% the location of the files to compile.
my_build_spec = [];
my_build_spec.rtwmakecfgDirs = {fullfile(pwd,'src')};

% Build the libraries in 'my_model'
rtw_precompile_libs(my_model,my_build_spec);
```

Input Arguments

`model` — Model object or name for which to build libraries
`object` | *`'modelName'`*

Name of the model containing the libraries that you want to build.

`build_spec` — Structure with field values that provides the build specification
struct

Structure with fields that define a build specification. Fields except `rtwmakecfgDirs` are optional.

Field Values in build_spec

Specify the structure field values of the `build_spec`.

Example: `build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};`

`rtwmakecfgDirs` — Fully qualified paths to the folders containing `rtwmakecfg` files for libraries to precompile
array of paths

Uses the `Name` and `Location` elements of `makeInfo.library`, as returned by the `rtwmakecfg` function, to specify name and location of precompiled libraries. If you use the `TargetPreCompLibLocation` parameter to specify the library folder, it overrides the `makeInfo.library.Location` setting.

The specified model must contain S-function blocks that use precompiled libraries, which the `rtwmakecfg` files specify. The makefile that the build approach generates contains the library rules only if the conversion requires the libraries.

Example: `build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};`

`libSuffix` — Suffix, including the file type extension, to append to the name of each library (for example, `_std.a` or `_vcx64.lib`)
character vector

The suffix must include a period (.). Set the suffix by using either this field or the `TargetLibSuffix` parameter. If you specify a suffix with both mechanisms, the `TargetLibSuffix` setting overrides the setting of this field.

Example: `build_spec.libSuffix = '_vcx64.lib';`

`intOnlyBuild` — Selects library optimization
`'false'` (default) | `'true'`

When set to `true`, indicates that the function optimizes the libraries so that they compile from integer code only. Applies to ERT-based targets only.

Example: `build_spec.intOnlyBuild = 'false';`

`makeOpts` — Specifies an option for `rtwMake`
character vector

Specifies an option to include in the `rtwMake` command line.

Example: `build_spec.makeOpts = '';`

`addLibs` — Specifies libraries to build
cell array of structures

This cell array of structures specifies the libraries to build that an `rtwmakecfg` function does not specify. Define each structure with two fields that are character arrays:

- `libName` — Name of the library without a suffix
- `libLoc` — Location for the precompiled library

The build approach (toolchain approach or template makefile approach) lets you specify other libraries and how to build them. Use this field if you must precompile libraries.

Example: `build_spec.addLibs = 'libs_list';`

See Also

Topics

“Precompile S-Function Libraries” (Simulink Coder)
“Recompile Precompiled Libraries” (Simulink Coder)
“Choose Build Approach and Configure Build Process” (Simulink Coder)
“Use rtwmakecfg.m API to Customize Generated Makefiles” (Simulink Coder)

Introduced in R2009b

rtwbuild

Build generated code from a model

Syntax

```
rtwbuild(model)
rtwbuild(model,name,value)

rtwbuild(subsystem)

rtwbuild(subsystem,'Mode','ExportFunctionCalls')
blockHandle = rtwbuild(subsystem,'Mode','ExportFunctionCalls')
```

Description

`rtwbuild(model)` generates code from `model` based on current model configuration parameter settings. If `model` is not already loaded into the MATLAB environment, `rtwbuild` loads it before generating code.

If you clear the **Generate code only** model configuration parameter, the function generates code and builds an executable image.

To reduce code generation time, when rebuilding a model, `rtwbuild` provides incremental model build. The code generator rebuilds a model or submodels only when they have changed since the most recent model build. To force a top-model build, see the `'ForceTopModelBuild'` argument.

Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (Parallel Computing Toolbox), for example, within a `parfor` or `spmd` loop. For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models” (Simulink Coder).

`rtwbuild(model,name,value)` uses additional options specified by one or more `name,value` pair arguments.

`rtwbuild(subsystem)` generates code from `subsystem` based on current model configuration parameter settings. Before initiating the build, open (or load) the parent model.

If you clear the **Generate code only** model configuration parameter, the function generates code and builds an executable image.

`rtwbuild(subsystem,'Mode','ExportFunctionCalls')` generates code from `subsystem` that includes function calls that you can export to external application code if you have Embedded Coder.

`blockHandle = rtwbuild(subsystem,'Mode','ExportFunctionCalls')` returns the handle to a SIL block created for code generated from the specified subsystem if **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** is set to `SIL` and if you have Embedded Coder. You can then use the SIL block for SIL verification testing.

Examples

Generate Code and Build Executable Image for Model

Generate C code for model `rtwdemo_rtwintro`.

```
rtwbuild('rtwdemo_rtwintro')
```

For the GRT system target file, the code generator produces the following code files and places them in folders `rtwdemo_rtwintro_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Files	Interface Files	Other Files

`rtwdemo_rtwintro.c`
`rtwdemo_rtwintro.h`
`rtwdemo_rtwintro_private.h`
`rtwdemo_rtwintrotypes.h`
	`rtGetInf.c`
	`rtGetInf.h`
	`rtGetNaN.c`
	`rtGetNaN.h`
	`rt_nonfinite.c`
	`rt_nonfinite.h`
	`rtwtypes.h`
	`multiword_types.h`
	`builtin_typeid_types.h`
		`rtmodel.h`	`rt_logging.c`

If the following model configuration parameters settings apply, the code generator produces additional results.

Parameter Setting	Results
Code Generation > **Generate code only** pane is cleared	Executable image `rtwdemo_rtwintro.exe`
Code Generation > **Report** > **Create code generation report** is selected	Report that provides information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Force Top Model Build

Generate code and build an executable image for `rtwdemo_mdlreftop`, which refers to model `rtwdemo_mdlrefbot`, regardless of model checksums and parameter settings.

```
rtwbuild('rtwdemo_mdlreftop', ...
    'ForceTopModelBuild',true)
```

Display Error Messages in Diagnostic Viewer

Introduce an error to model `rtwdemo_mdlreftop` and save the model as `rtwdemo_mdlreftop_witherr`. Display build error messages in the Diagnostic Viewer and in the Command Window while generating code and building an executable image for model `rtwdemo_mdlreftop_witherr`.

```
rtwbuild('rtwdemo_mdlreftop_witherr', ...
    'OkayToPushNags',true)
```

Generate Code and Build Executable Image for Subsystem

Generate C code for subsystem `Amplifier` in model `rtwdemo_rtwintro`.

```
rtwbuild('rtwdemo_rtwintro/Amplifier')
```

For the GRT target, the code generator produces the following code files and places them in folders `Amplifier_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Files	Interface Files	Other Files
`Amplifier.c`	`rtGetInf.c`	`rtmodel.h`	`rt_logging.c`
`Amplifier.h`	`rtGetInf.h`		
`Amplifier_private.h`	`rtGetNaN.c`		
`Amplifier_types.h`	`rtGetNaN.h`		
	`rt_nonfinite.c`		
	`rt_nonfinite.h`		
	`rtwtypes.h`		
	`multiword_types.h`		
	`builtin_typeid_types.h`		

If you apply the parameter settings listed in the table, the code generator produces the results listed.

Parameter Setting	Results
Code Generation > **Generate code only** pane is cleared	Executable image `Amplifier.exe`
Code Generation > **Report** > **Create code generation report** is selected	Report that provides information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Build Subsystem for Exporting Code to External Application

To export the image to external application code, build an executable image from a function-call subsystem.

```
rtwdemo_exporting_functions
rtwbuild('rtwdemo_exporting_functions/rtwdemo_subsystem','Mode','ExportFunctionCalls')
```

The executable image `rtwdemo_subsystem.exe` appears in your working folder.

Create SIL Block for Verification

From a function-call subsystem, create a SIL block that you can use to test the code generated from a model.

Open subsystem `rtwdemo_subsystem` in model `rtwdemo_exporting_functions` and set **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** to `SIL`.

Create the SIL block.

```
mysilblockhandle = rtwbuild('rtwdemo_exporting_functions/rtwdemo_subsystem',...
'Mode','ExportFunctionCalls')
```

The code generator produces a SIL block for the generated subsystem code. You can add the block to an environment or test harness model that supplies test vectors or stimulus input. You can then run simulations that perform SIL tests and verify that the generated code in the SIL block produces the same result as the original subsystem.

Name Exported Initialization Function

Name the initialization function generated when building an executable image from a function-call subsystem.

```
rtwdemo_exporting_functions
rtwbuild('rtwdemo_exporting_functions/rtwdemo_subsystem',...
'Mode','ExportFunctionCalls','ExportFunctionInitializeFunctionName','subsysinit')
```

The initialization function name `subsysinit` appears in `rtwdemo_subsystem_ert_rtw/ert_main.c`.

Display Status Information in Build Process Status Window

Display build information in the Build Process Status Window while generating code and running a parallel build of model `rtwdemo_mdlreftop_witherr`.

```
rtwbuild('rtwdemo_mdlreftop_witherr', ...
    'OpenBuildStatusAutomatically',true)
```

Input Arguments

`model` — Model object or name for which to generate code or build an executable image

`object` | *`'modelName'`*

Model for which to generate code or build an executable image, specified as an object or a character vector representing the model name.

Example: `'rtwdemo_exporting_functions'`

`subsystem` — Subsystem name for which to generate code or build executable image

`'subsystemName'`

Subsystem for which to generate code or build an executable image, specified as a character vector representing the subsystem name or the full block path.

Example: `'rtwdemo_exporting_functions/rtwdemo_subsystem'`

`name,value` — Name-value pairs select options for the build process

name-value pairs

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `rtwbuild('rtwdemo_mdlreftop','ForceTopModelBuild',true)`

`ForceTopModelBuild` — Force regeneration of top model code
`false` (default) | `true`

Force regeneration of top model code, specified as `true` or `false`.

Action	Specify
Force the code generator to regenerate code for the top model of a system that includes referenced models	`true`
Specify that the code generator determine whether to regenerate top model code based on model and model parameter changes	`false`

Consider forcing regeneration of code for a top model if you change items associated with external or custom code, such as code for a custom target. For example, set `ForceTopModelBuild` to `true` if you change:

- TLC code
- S-function source code, including `rtwmakecfg.m` files
- Integrated custom code

Alternatively, you can force regeneration of top model code by deleting folders in the code generation folder (Simulink), such as `slprj` or the generated model code folder.

`OkayToPushNags` — Display build error messages in Diagnostic Viewer
`false` (default) | `true`

Display error messages from the build in Diagnostic Viewer, specified as `true` or `false`.

Action	Specify
Display build error messages in the Diagnostic Viewer and in the Command Window	`true`

Action	Specify
Display build error messages in the Command Window only	`false`

`generateCodeOnly` — Generate code only
`false` | `true`

If you do not specify a value, the **Generate code only** (`GenCodeOnly`) option on the **Code Generation** pane controls build process behavior.

If you specify a value, the argument overrides the **Generate code only** (`GenCodeOnly`) option on the **Code Generation** pane.

Action	Specify
Generate code only.	`true`
Generate code and build executable file.	`false`

`Mode` — (for subsystem builds only) Direct code generator to export function calls
`'ExportFunctionCalls'` (default)

If you have Embedded Coder, generates code from `subsystem` that includes function calls that you can export to external application code.

`ExportFunctionInitializeFunctionName` — Function name
character vector

Name the exported initialization function for specified subsystem.

Example:
`rtwbuild(`*`subsystem`*`,'Mode','ExportFunctionCalls','ExportFunctionInitializeFunctionName',` *`fcnname`*`)`

`OpenBuildStatusAutomatically` — Display build information in the Build Process Status Window
`false` (default) | `true`

Display build information in the **Build Process Status** window, specified as `true` or `false`. For more information about using the status window, see “View Build Process Status” (Simulink Coder).

The **Build Process Status** window support parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

Action	Specify
Display build information in the Build Process Status Window	`true`
No action	`false`

`ObfuscateCode` — Generate obfuscated C code
false (default) | true

Specify whether to generate obfuscated C code, specified as `true` or `false`.

Action	Specify
Generate obfuscated C code that you can share with third parties with reduced likelihood of compromising intellectual property.	`true`
No action.	`false`

Output Arguments

`blockHandle` — Handle to SIL block created for generated subsystem code
handle

Handle to SIL block created for generated subsystem code. Returned only if both of the following conditions apply:

- You are licensed to use Embedded Coder software.
- **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** is set to `SIL`.

Tips

You can initiate code generation and the build process by:

- Pressing **Ctrl+B**.
- Selecting **Code** > **C/C++ Code** > **Build Model**.

- Invoking the `slbuild` command from the MATLAB command line.

See Also

`coder.buildstatus.close` | `coder.buildstatus.open` | `rtwrebuild` | `slbuild`

Topics

“Build and Run a Program” (Simulink Coder)
“Choose Build Approach and Configure Build Process” (Simulink Coder)
“Control Regeneration of Top Model Code” (Simulink Coder)
“Generate Component Source Code for Export to External Code Base”
“Software-in-the-Loop Simulation”

Introduced in R2009a

RTW.getBuildDir

Get build folder information from model build information

Syntax

```
RTW.getBuildDir(model)
folderStruct = RTW.getBuildDir(model)
```

Description

`RTW.getBuildDir(model)` displays build folder information for model.

If the model is closed, the function opens and then closes the model, leaving it in its original state. If the model is large and closed, the `RTW.getBuildDir` function can take longer to execute.

`folderStruct = RTW.getBuildDir(model)` returns a structure containing build folder information.

You can use this function in automated scripts to determine the build folder in which the generated code for a model is placed.

This function can return build folder information for protected models.

Examples

Display Build Folder Information

Display build folder information for the model `'sldemo_fuelsys'`.

```
>> RTW.getBuildDir('sldemo_fuelsys')


ans =
```

```
              BuildDirectory: 'C:\work\modelref\sldemo_fuelsys_ert_rtw'
                 CacheFolder: 'C:\work\modelref'
                CodeGenFolder: 'C:\work\modelref'
            RelativeBuildDir: 'sldemo_fuelsys_ert_rtw'
              BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\sldemo_fuelsys'
      ModelRefRelativeSimDir: 'slprj\sim\sldemo_fuelsys'
      ModelRefRelativeHdlDir: 'slprj\hdl\sldemo_fuelsys'
           ModelRefDirSuffix: ''
           SharedUtilsSimDir: 'slprj\sim\_sharedutils'
           SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

Get Build Folder Information

Return a structure `my_folderStruct` that contains build folder information for the model `'MyModel'`.

```
>> my_folderStruct = RTW.getBuildDir('MyModel')


my_folderStruct =

              BuildDirectory: 'H:\MyModel_ert_rtw'
                 CacheFolder: 'H:\'
                CodeGenFolder: 'H:\'
            RelativeBuildDir: 'MyModel_ert_rtw'
              BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\MyModel'
      ModelRefRelativeSimDir: 'slprj\sim\MyModel'
      ModelRefRelativeHdlDir: 'slprj\hdl\MyModel'
           ModelRefDirSuffix: ''
```

```
SharedUtilsSimDir: 'slprj\sim\_sharedutils'
SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

Input Arguments

`model` — Model object or name for which to get the build folders

`object` | *`'modelName'`*

Model for which to get the build folder, specified as an object or a character vector representing the model name.

Example: `'sldemo_fuelsys'`

Output Arguments

`folderStruct` — Structure with field values that provide build folder information

struct

Structure with fields that provides build folder information.

Example: folderstruct = RTW.getBuildDir('MyModel')

`BuildDirectory` — Character vector specifying fully qualified path to build folder for model

character vector

`CacheFolder` — Character vector specifying root folder in which to place model build artifacts used for simulation

character vector

`CodeGenFolder` — Character vector specifying root folder in which to place code generation files

character vector

`RelativeBuildDir` — Character vector specifying build folder relative to the current working folder (`pwd`)

character vector

`BuildDirSuffix` — Character vector specifying suffix appended to model name to create build folder
character vector

`ModelRefRelativeRootSimDir` — Character vector specifying the relative root folder for the model reference target simulation folder
character vector

`ModelRefRelativeRootTgtDir` — Character vector specifying the relative root folder for the model reference target build folder
character vector

`ModelRefRelativeBuildDir` — Character vector specifying model reference target build folder relative to current working folder (`pwd`)
character vector

`ModelRefRelativeSimDir` — Character vector specifying model reference target simulation folder relative to current working folder (`pwd`)
character vector

`ModelRefRelativeHdlDir` — Character vector specifying model reference target HDL folder relative to current working folder (`pwd`)
character vector

`ModelRefDirSuffix` — Character vector specifying suffix appended to system target file name to create model reference build folder
character vector

`SharedUtilsSimDir` — Character vector specifying the shared utility folder for simulation
character vector

`SharedUtilsTgtDir` — Character vector specifying the shared utility folder for code generation
character vector

See Also

`rtwbuild`

Topics

“Working Folder” (Simulink Coder)
“Manage Build Process Folders” (Simulink Coder)

Introduced in R2008b

rtwrebuild

Rebuild generated code from model

Syntax

```
rtwrebuild()

rtwrebuild(model)

rtwrebuild(path)
```

Description

`rtwrebuild()` assumes that the current working folder is the build folder of the model (not the model location) and invokes the makefile in the build folder. If the current working folder is not the build folder, the function exits with an error.

`rtwrebuild` invokes the makefile generated during the previous build to recompile files you modified since that build. Operation of this function depends on the current working folder, not the current loaded model. If your model includes referenced models, `rtwrebuild` invokes the makefile for referenced model code recursively before recompiling the top model.

Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (Parallel Computing Toolbox), for example, within a `parfor` or `spmd` loop. For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models” (Simulink Coder).

`rtwrebuild(model)` assumes that the current working folder is one level above the build folder and invokes the makefile in the build folder. If the current working folder (`pwd`) is not one level above the build folder, the function exits with an error.

`rtwrebuild(path)` finds the build folder indicated with the *`path`* argument and invokes the makefile in the build folder. The *`path`* argument syntax lets the function operate without regard to the relationship between the current working folder and the build folder of the model.

Examples

Rebuild Code from Build Folder

Invoke the makefile and recompile code when the current working folder is the build folder. For example,

- If the model name is `mymodel`
- And, if the model build was initiated in the `C:\work` folder
- And, if the system target is GRT

Invoke the previously generated makefile in the current working folder (build folder) `C:\work\mymodel_grt_rtw`.

```
rtwrebuild()
```

Rebuild Code from Parent Folder of Build Folder

When the current working folder is one level above the build folder, invoke the makefile and recompile code.

```
rtwrebuild('mymodel')
```

Rebuild Code from a Folder

Invoke the makefile and recompile code from a current folder by specifying a path to the model build folder, `C:\work\mymodel_grt_rtw`.

```
rtwrebuild(fullfile('C:','work','mymodel_grt_rtw'))
```

Input Arguments

`model` — Model object or name for which to regenerate code or rebuild an executable image
`object` | `'modelName'`

Model for which to regenerate code or rebuild an executable image, specified as an object or a character vector representing the model name.

Example: `'rtwdemo_exporting_functions'`

path — Model path object or fully qualified path to the build folder for the model for which to regenerate code or rebuild an executable image
`object` | *`modelPath`*

Example: `fullfile('C:','work','mymodel_grt_rtw')`

See Also

`rtwbuild` | `slbuild`

Topics

“Rebuild a Model” (Simulink Coder)

Introduced in R2009a

rtwreport

Create generated code report for model with Simulink Report Generator

Syntax

```
rtwreport(model)
rtwreport(model,folder)
```

Description

`rtwreport(model)` creates a report of code generation information for a model. Before creating the report, the function loads the model and generates code. The code generator names the report `codegen.html`. It places the file in your current folder. The report includes:

- Snapshots of the model, including subsystems.
- Block execution order list.
- Code generation summary with a list of generated code files, configuration settings, a subsystem map, and a traceability report.
- Full listings of generated code that reside in the build folder.

`rtwreport(model,folder)` specifies the build folder, *`model_target`*`_rtw`. The build folder (`folder`) and `slprj` folder must reside in the code generation folder (Simulink). If the software cannot find the `folder`, an error occurs and code is not generated.

Examples

Create Report Documenting Generated Code

Create a report for model `rtwdemo_secondOrderSystem`:

```
rtwreport('rtwdemo_secondOrderSystem');
```

Create Report Specifying Build Folder

Create a report for model `rtwdemo_secondOrderSystem` using build folder, `rtwdemo_secondOrderSystem_grt_rtw`:

```
rtwreport('rtwdemo_secondOrderSystem', ...
    'rtwdemo_secondOrderSystem_grt_rtw');
```

Input Arguments

`model` — Model name
character vector

Model name for which the report is generated, specified as a character vector.

Example: `'rtwdemo_secondOrderSystem'`

Data Types: `char`

`folder` — Build folder name
character vector

Build folder name, specified as a character vector. When you have multiple build folders, include a folder name. For example, if you have multiple builds using different targets, such as GRT and ERT.

Example: `'rtwdemo_secondOrderSystem_grt_rtw'`

Data Types: `char`

See Also

Topics

“Document Generated Code with Simulink Report Generator” (Simulink Coder)
`Import Generated Code`
“Working with the Report Explorer” (Simulink Report Generator)
`Code Generation Summary`

Introduced in R2007a

rtwtrace

Trace a block to generated code in HTML code generation report

Syntax

```
rtwtrace('blockpath')
rtwtrace('Simulink_identifier')
rtwtrace('blockpath', 'hdl')
rtwtrace('blockpath', 'plc')
```

Description

`rtwtrace('blockpath')` opens an HTML code generation report that displays contents of the source code file and highlights the line of code corresponding to the specified block.

Before calling `rtwtrace`, make sure that:

- You select an ERT-based model and enable model to code navigation.

  In the Configuration Parameters dialog box, select the **Model-to-code** (Simulink Coder) parameter.
- You generate code for the model by using the code generator.
- Your build folder is under the current working folder. Otherwise, `rtwtrace` might produce an error.

`rtwtrace('Simulink_identifier')` opens an HTML code generation report that displays contents of the source code file and highlights the line of code corresponding to the block identified by the Simulink identifier (SID). SID is a unique designation for each block or element in the model. For more information, see “Locate Diagram Components Using Simulink Identifiers” (Simulink).

`rtwtrace('blockpath', 'hdl')` opens an HTML code generation report in HDL Coder™ that displays contents of the source code file and highlights the line of code corresponding to the specified block.

`rtwtrace('blockpath', 'plc')` opens an HTML code generation report in Simulink PLC Coder™ that displays contents of the source code file and highlights the line of code corresponding to the specified block.

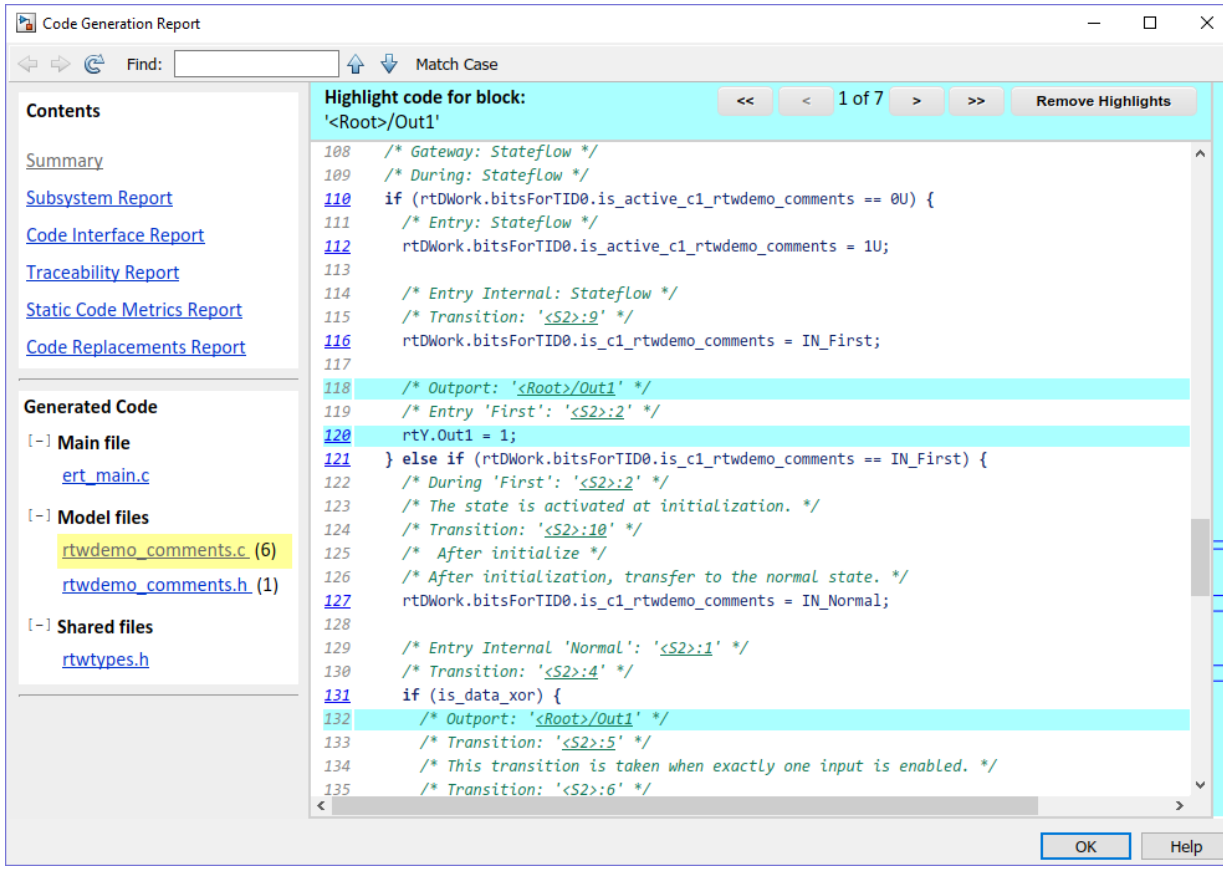
Examples

Display Generated Code for a Block

Display the generated code for block `Out1` in the model `rtwdemo_comments` in HTML code generation report:

```
% Using block path
rtwtrace('rtwdemo_comments/Out1')

% Using Simulink identifier
rtwtrace('rtwdemo_comments:33')
```



Input Arguments

`blockpath` — block path

character vector (default)

`blockpath` is a character vector enclosed in quotes specifying the full Simulink block path, for example, '*`model_name/block_name`*'.

Example: `'rtwdemo_comments/Out1'`

Data Types: `char`

`Simulink_identifier` — Simulink identifier
character vector (default)

`Simulink_identifier` is a character vector enclosed in quotes specifying the Simulink identifier, for example, `'model_name:number'`.

Example: `'rtwdemo_comments:33'`

Data Types: `char`

`hdl` — HDL Coder
character vector

`hdl` is a character vector enclosed in quotes specifying that the code report is from HDL Coder.

Example: `'Out1'`

Data Types: `char`

`plc` — PLC Coder
character vector

`plc` is a character vector enclosed in quotes specifying that the code report is from Simulink PLC Coder.

Example: `'Out1'`

Data Types: `char`

Alternatives

To trace from a block in the model diagram, right-click a block and select **C/C++ Code** > **Navigate to C/C++ Code**.

See Also

Topics

“Model-to-Code Traceability”
“Model-to-code” (Simulink Coder)

Introduced in R2009b

setTargetProvidesMain

Disable inclusion of code generator provided (generated or static) `main.c` source file during model build

Syntax

```
setTargetProvidesMain(buildinfo,providesmain)
```

Description

`setTargetProvidesMain(buildinfo,providesmain)` disables the code generator from including a sample `main.c` source file.

To replace the sample `main.c` file from the code generator with a custom `main.c` file, call the `setTargetProvidesMain` function during the `'after_tlc'` case in the `ert_make_rtw_hook.m` or `grt_make_rtw_hook.m` file.

Examples

Workflow for `setTargetProvidesMain`

To apply the `setTargetProvidesMain` function:

Add `buildInfo` to the arguments in the function call.

```
function ert_make_rtw_hook(hookMethod,modelName,rtwroot, ...
   templateMakefile,buildOpts,buildArgs,buildInfo)
```

Add the `setTargetProvidesMain` function to the `'after_tlc'` stage.

```
case 'after_tlc'
 % Called just after to invoking TLC Compiler (actual code generation.)
 % Valid arguments at this stage are hookMethod, modelName, and
 % buildArgs, buildInfo
```

```
%
setTargetProvidesMain(buildInfo,true);
```

Use the **Configuration Parameters > Code Generation > Custom Code > Source Files** field to add your custom `main.c` to the model. When you indicate that the target provides `main.c`, the model requires this file to build without errors.

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`providesmain` — Logical value that specifies whether the code generator includes the target provided `main.c` file
false (default) | true

The *`providesmain`* argument specifies whether the code generator includes a (generated or static) `main.c` source file.

- `false` — The code generator includes a sample `main.obj` object file.
- `true` — The target provides the `main.c` source file.

See Also
`addSourceFiles` | `addSourcePaths`

Topics
“Customize Build Process with STF_make_rtw_hook File” (Simulink Coder)

Introduced in R2009a

Simulink.fileGenControl

Specify root folders for files generated by diagram updates and model builds

Syntax

```
cfg = Simulink.fileGenControl('getConfig')
Simulink.fileGenControl(Action,Name,Value)
```

Description

`cfg = Simulink.fileGenControl('getConfig')` returns a handle to an instance of the `Simulink.FileGenConfig` object, which contains the current values of these file generation control parameters:

- `CacheFolder` – Specifies the root folder for model build artifacts that are used for simulation, including Simulink® cache files.
- `CodeGenFolder` – Specifies the root folder for code generation files.
- `CodeGenFolderStructure` – Controls the folder structure within the code generation folder.

To get or set the parameter values, use the `Simulink.FileGenConfig` object.

These Simulink preferences determine the initial parameter values for the MATLAB session:

- Simulation cache folder (Simulink) – `CacheFolder`
- Code generation folder (Simulink) – `CodeGenFolder`
- Code generation folder structure (Simulink) – `CodeGenFolderStructure`

`Simulink.fileGenControl(Action,Name,Value)` performs an action that uses the file generation control parameters of the current MATLAB session. Specify additional options with one or more `name,value` pair arguments.

Examples

Get File Generation Control Parameter Values

To obtain the file generation control parameter values for the current MATLAB session, use `getConfig`.

```
cfg = Simulink.fileGenControl('getConfig');

myCacheFolder = cfg.CacheFolder;
myCodeGenFolder = cfg.CodeGenFolder;
myCodeGenFolderStructure = cfg.CodeGenFolderStructure;
```

Set File Generation Control Parameters by Using `Simulink.FileGenConfig` Object

To set the file generation control parameter values for the current MATLAB session, use the `setConfig` action. First, set values in an instance of the `Simulink.FileGenConfig` object. Then, pass the object instance. This example assumes that your system has `aNonDefaultCacheFolder` and `aNonDefaultCodeGenFolder` folders.

```
% Get the current configuration
cfg = Simulink.fileGenControl('getConfig');

% Change the parameters to non-default locations
% for the cache and code generation folders
cfg.CacheFolder = fullfile('C:','aNonDefaultCacheFolder');
cfg.CodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');
cfg.CodeGenFolderStructure = 'TargetEnvironmentSubfolder';

Simulink.fileGenControl('setConfig', 'config', cfg);
```

Set File Generation Control Parameters Directly

You can set file generation control parameter values for the current MATLAB session without creating an instance of the `Simulink.FileGenConfig` object. This example assumes that your system has `aNonDefaultCacheFolder` and `aNonDefaultCodeGenFolder` folders.

```
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ...
   'CodeGenFolder', myCodeGenFolder, ...
   'CodeGenFolderStructure', ...
    Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);
```

If you do not want to generate code for different target environments in separate folders, for `'CodeGenFolderStructure'`, specify the value `Simulink.filegen.CodeGenFolderStructure.ModelSpecific`.

Reset File Generation Control Parameters

You can reset the file generation control parameters to values from Simulink preferences.

```
Simulink.fileGenControl('reset');
```

Create Simulation Cache and Code Generation Folders

To create file generation folders, use the `set` action with the `'createDir'` option. You can keep previous file generation folders on the MATLAB path through the `'keepPreviousPath'` option.

```
%
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', ...
   'CacheFolder',myCacheFolder, ...
   'CodeGenFolder',myCodeGenFolder, ...
   'keepPreviousPath',true, ...
   'createDir',true);
```

Input Arguments

`Action` — Specify action

`'reset'` | `'set'` | `'setConfig'`

Specify an action that uses the file generation control parameters of the current MATLAB session:

- `'reset'` – Reset file generation control parameters to values from Simulink preferences.
- `'set'` – Set file generation control parameters for the current MATLAB session by directly passing values.
- `'setConfig'` – Set file generation control parameters for the current MATLAB session by using an instance of a `Simulink.FileGenConfig` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Simulink.fileGenControl(`*`Action`*`, `*`Name`*`, `*`Value`*`);`

`config` — Specify instance of `Simulink.FileGenConfig`
object handle

Specify the `Simulink.FileGenConfig` object instance containing file generation control parameters that you want to set.

Option for `setConfig`.

Example: `Simulink.fileGenControl('setConfig', 'config', `*`cfg`*`);`

`CacheFolder` — Specify simulation cache folder
character vector

Specify a simulation cache folder path value for the `CacheFolder` parameter.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', `*`myCacheFolder`*`);`

`CodeGenFolder` — Specify code generation folder
character vector

Specify a code generation folder path value for the `CodeGenFolder` parameter. You can specify an absolute path or a path relative to build folders. For example:

- `'C:\Work\mymodelsimcache'` and `'/mywork/mymodelgencode'` specify absolute paths.
- `'mymodelsimcache'` is a path relative to the current working folder (`pwd`). The software converts a relative path to a fully qualified path at the time the `CacheFolder` or `CodeGenFolder` parameter is set. For example, if `pwd` is `'/mywork'`, the result is `'/mywork/mymodelsimcache'`.
- `'../test/mymodelgencode'` is a path relative to `pwd`. If `pwd` is `'/mywork'`, the result is `'/test/mymodelgencode'`.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CodeGenFolder', myCodeGenFolder);`

`CodeGenFolderStructure` — Specify generated code folder structure
`Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) | `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder`

Specify the layout of subfolders within the generated code folder:

- `Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) – Place generated code in subfolders within a model-specific folder.
- `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder` – If models are configured for different target environments, place generated code for each model in a separate subfolder. The name of the subfolder corresponds to the target environment.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ... 'CodeGenFolder', myCodeGenFolder, ... 'CodeGenFolderStructure', ... Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);`

`keepPreviousPath` — Keep previous folder paths on MATLAB path
`false` (default) | `true`

Specify whether to keep the previous values of `CacheFolder` and `CodeGenFolder` on the MATLAB path:

- `true` – Keep previous folder path values on MATLAB path.

- `false` (default) – Remove previous older path values from MATLAB path.

Option for `reset`, `set`, or `setConfig`.

Example: `Simulink.fileGenControl('reset', 'keepPreviousPath', true);`

`createDir` — Create folders for file generation
false (default) | true

Specify whether to create folders for file generation if the folders do not exist:

- `true` – Create folders for file generation.
- `false` (default) – Do not create folders for file generation.

Option for `set` or `setConfig`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, 'CodeGenFolder', myCodeGenFolder, 'keepPreviousPath', true, 'createDir',true);`

Avoid Naming Conflicts

Using `Simulink.fileGenControl` to set `CacheFolder` and `CodeGenFolder` adds the specified folders to your MATLAB search path. This function has the same potential for introducing a naming conflict as using `addpath` to add folders to the search path. For example, a naming conflict occurs if the folder that you specify for `CacheFolder` or `CodeGenFolder` contains a model file with the same name as an open model. For more information, see “What Is the MATLAB Search Path?” (MATLAB) and “Files and Folders that MATLAB Accesses” (MATLAB).

To use a nondefault location for the simulation cache folder or code generation folder:

1. Delete any potentially conflicting artifacts that exist in:
   - The current working folder, `pwd`.
   - The nondefault simulation cache and code generation folders that you intend to use.
2. Specify the nondefault locations for the simulation cache and code generation folders by using `Simulink.fileGenControl` or Simulink preferences.

Output Arguments

`cfg` — Current values of file generation control parameters
object handle

Instance of a `Simulink.FileGenConfig` object, which contains the current values of file generation control parameters.

See Also

“Simulation cache folder” (Simulink) | “Code generation folder” (Simulink) | Code generation folder structure (Simulink)

Topics

“Manage Build Process Folders” (Simulink Coder)
“Share Simulation Builds for Faster Simulations” (Simulink)

Introduced in R2010b

Simulink.ModelReference.modifyProtectedModel

Modify existing protected model

Syntax

```
Simulink.ModelReference.modifyProtectedModel(model)
Simulink.ModelReference.modifyProtectedModel(model,Name,Value)

[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(
model,'Harness',true)
[~ ,neededVars] = Simulink.ModelReference.modifyProtectedModel(
model)
```

Description

`Simulink.ModelReference.modifyProtectedModel(model)` modifies options for an existing protected model created from the specified `model`. If `Name,Value` pair arguments are not specified, the modified protected model is updated with default values and supports only simulation.

`Simulink.ModelReference.modifyProtectedModel(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. These options are the same options that are provided by the `Simulink.ModelReference.protect` function. However, these options have additional options to change encryption passwords for read-only view, simulation, and code generation. When you add functionality to the protected model or change encryption passwords, the unprotected model must be available. The software searches for the model on the MATLAB path. If the model is not found, the software reports an error.

`[harnessHandle] = Simulink.ModelReference.modifyProtectedModel( model,'Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~ ,neededVars] = Simulink.ModelReference.modifyProtectedModel( model)` returns a cell array that includes the names of base workspace variables used by the protected model.

Examples

Update Protected Model with Default Values

Create a modifiable protected model with support for code generation, then reset it to default values.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view. Optionally, if you want to add support for HDL code generation, set 'hdl' to true.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Modify the model to use default values.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter');
```

The resulting protected model is updated with default values and supports only simulation.

Remove Functionality from Protected Model

Create a modifiable protected model with support for code generation and Web view, then modify it to remove the Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Webview',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Remove support for Web view from the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter', 'Mode', 'CodeGeneration','Report',true);
```

Change Encryption Password for Code Generation

Change an encryption password for a modifiable protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add the password that the protected model user must provide to generate code.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','cgpassword');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Encrypt',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Change the encryption password for simulation.

```
Simulink.ModelReference.modifyProtectedModel(
'sldemo_mdlref_counter','Mode','CodeGeneration','Encrypt',true,...
'Report',true,'ChangeSimulationPassword',...
{'cgpassword','new_password'});
```

Add Harness Model for Protected Model

Add a harness model for an existing protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with a report and support for code generation with encryption. Optionally, if you want to add support for HDL code generation, set `'hdl'` to `true`.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add a harness model for the protected model.

```
[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Report',true,...
'Harness',true);
```

Input Arguments

`model` — Model name

string or character vector (default)

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

General

`Path` — Folder for protected model
current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.

Example: `'Path','C:\Work'`

`Report` — Option to generate a report
false (default) | true

Option to generate a report, specified as a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: `'Report',true`

`hdl` — Option to generate HDL code
false (default) | true

Option to generate HDL code, specified as a Boolean value.

This option requires HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: `'hdl',true`

`Harness` — Option to create a harness model

false (default) | true

Option to create a harness model, specified as a Boolean value.

Example: `'Harness',true`

`CustomPostProcessingHook` — Option to add postprocessing function for protected model files

function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. The object also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example: `'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)`

Functionality

`Mode` — Model protection mode

`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'HDLCodeGeneration'` | `'ViewOnly'`

Model protection mode. Specify one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode.
- `'CodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support code generation.
- `'HDLCodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support HDL code generation.
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. Turns on the read-only view mode.

Example: `'Mode','Accelerator'`

`OutputFormat` — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

Note This argument affects the output only when you specify `Mode` as `'Accelerator'` or `'CodeGeneration`. When you specify `Mode` as `'Normal'`, only a MEX-file is part of the output package.

Protected code visibility. This argument determines what part of the code generated for a protected model is visible to users. Specify one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. Code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. Code in the build folder is visible. Header files referenced by the code are also visible.

Example: `'OutputFormat','AllReferencedHeaders'`

`ObfuscateCode` — Option to obfuscate generated code
true (default) | false

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation is enabled for the protected model. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode',true`

`Webview` — Option to include a Web view

false (default) | true

Option to include a read-only view of protected model, specified as a Boolean value.

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdlref_counter`, you can call:

  ```
  Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');
  ```

- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

Encryption

`ChangeSimulationPassword` — Option to change the encryption password for simulation

cell array of two character vectors

Option to change the encryption password for simulation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeSimulationPassword',{'old_password','new_password'}`

`ChangeViewPassword` — Option to change the encryption password for read-only view

cell array of two character vectors

Option to change the encryption password for read-only view, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeViewPassword',{'old_password','new_password'}`

`ChangeCodeGenerationPassword` — Option to change the encryption password for code generation
cell array of two character vectors

Option to change the encryption password for code generation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeCodeGenerationPassword', {'old_password','new_password'}`

`Encrypt` — Option to encrypt protected model
false (default) | true

Option to encrypt a protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model: `Simulink.ModelReference.ProtectedModel.setPasswordForView`
- Password for simulation: `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`
- Password for code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`
- Password for HDL code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration`

Example: `'Encrypt',true`

Output Arguments

`harnessHandle` — Handle of the harness model
double

Handle of the harness model, returned as a double or `0`, depending on the value of `Harness`.

If `Harness` is `true`, the value is the handle of the harness model; otherwise, the value is `0`.

`neededVars` — Names of base workspace variables
cell array

Names of base workspace variables used by the protected model, returned as a cell array.

The cell array can also include variables that the protected model does not use.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForModify` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.protect

Obscure referenced model contents to hide intellectual property

Syntax

```
Simulink.ModelReference.protect(model)
Simulink.ModelReference.protect(model,Name,Value)

[harnessHandle] = Simulink.ModelReference.protect(model,'
Harness',true)
[~ ,neededVars] = Simulink.ModelReference.protect(model)
```

Description

`Simulink.ModelReference.protect(model)` creates a protected model from the specified `model`. It places the protected model in the current working folder. The protected model has the same name as the source model. It has the extension `.slxp`.

`Simulink.ModelReference.protect(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[harnessHandle] = Simulink.ModelReference.protect(model,'Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~ ,neededVars] = Simulink.ModelReference.protect(model)` returns a cell array that includes the names of base workspace variables used by the protected model.

Examples

Protect Referenced Model

Protect a referenced model and place the protected model in the current working folder.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the current working folder.

Place Protected Model in Specified Folder

Protect a referenced model and place the protected model in a specified folder.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Path','C:\Work');
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in `C:\Work`.

Generate Code for Protected Model

Protect a referenced model, generate code for it in normal mode, and obfuscate the code.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Path','C:\Work','Mode','CodeGeneration',...
'ObfuscateCode',true);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the `C:\Work` folder. The protected model runs as a child of the parent model. The code generated for the protected model is obfuscated by the software.

Generate HDL Code for Protected Model

Protect a referenced model, and generate HDL code for it in normal mode.

```
parent_model= 'hdlcoder_protected_model_parent_harness';
```

```
reference_model_to_protect = 'hdlcoder_referenced_model_gain';

Simulink.ModelReference.protect(reference_model_to_protect, ...
                        'Mode','HDLCodeGeneration')
```

A protected model named `hdlcoder_referenced_model_gain.slxp` is created. The protected model file is placed in the same folder as the parent model and the referenced model. The protected model runs as a child of the parent model.

Set the **hdl** option to `true` with **Mode** set to `CodeGeneration` to enable both C code generation and HDL code generation support for a protected model that you create.

```
parent_model= 'hdlcoder_protected_model_parent_harness';
reference_model_to_protect = 'hdlcoder_referenced_model_gain';

Simulink.ModelReference.protect(reference_model_to_protect, ...
                        'Mode','CodeGeneration','hdl',true)
```

Control Code Visibility for Protected Model

Control code visibility by allowing users to view only binary files and headers in the code generated for a protected model.

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Mode','CodeGeneration','OutputFormat',...
'CompiledBinaries');
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the current working folder. Users can view only binary files and headers in the code generated for the protected model.

Create Harness Model for Protected Model

Create a harness model for a protected model and generate an HTML report.

```
sldemo_mdlref_bus;
modelPath= 'sldemo_mdlref_bus/CounterA'

[harnessHandle] = Simulink.ModelReference.protect(modelPath,'Path','C:\Work',...
'Harness',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created, along with an untitled harness model. The protected model file is placed in the `C:\Work` folder. The

folder also contains an HTML report. The handle of the harness model is returned in `harnessHandle`.

Input Arguments

`model` — Model name
string or character vector (default)

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the model to be protected.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

`Harness` — Option to create a harness model
false (default) | true

Option to create a harness model, specified as a Boolean value.

Example: `'Harness',true`

`Mode` — Model protection mode
`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'HDLCodeGeneration'` | `'ViewOnly'`

Model protection mode. Specify one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode.

- `'CodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support code generation.
- `'HDLCodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support HDL code generation. (Requires HDL Coder license)
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. Turns on the read-only view mode.

Example: `'Mode','Accelerator'`

`CodeInterface` — Interface through which generated code is accessed by Model block
`'Model reference'` (default) | `'Top model'`

Applies only if the system target file (`SystemTargetFile`) is set to an ERT-based system target file (for example, `ert.tlc`). Requires Embedded Coder license.

Specify one of the following values:

- `'Model reference'`: Code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.
- `'Top model'`: Code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

Example: `'CodeInterface','Top model'`

`ObfuscateCode` — Option to obfuscate generated code
true (default) | false

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation during protection is enabled. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode',true`

`Path` — Folder for protected model
current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.

Example: `'Path','C:\Work'`

`Report` — Option to generate a report
false (default) | true

Option to generate a report, specified as a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: `'Report',true`

`hdl` — Option to generate HDL code
false (default) | true

Option to generate HDL code, specified as a Boolean value.

This option requires HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: `'hdl',true`

`OutputFormat` — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

Note This argument affects the output only when you specify `Mode` as `'Accelerator'` or `'CodeGeneration`. When you specify `Mode` as `'Normal'`, only a MEX-file is part of the output package.

Protected code visibility. This argument determines what part of the code generated for a protected model is visible to users. Specify one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. All code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. All code in the build folder is visible. All headers referenced by the code are also visible.

Example: `'OutputFormat','AllReferencedHeaders'`

`Webview` — Option to include a Web view
false (default) | true

Option to include a read-only view of protected model, specified as a Boolean value.

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdlref_counter`, you can call:

  ```
  Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');
  ```
- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

`Encrypt` — Option to encrypt protected model
false (default) | true

Option to encrypt a protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model: `Simulink.ModelReference.ProtectedModel.setPasswordForView`
- Password for simulation: `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`
- Password for code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`

- Password for HDL code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration`

Example: `'Encrypt',true`

`CustomPostProcessingHook` — Option to add postprocessing function for protected model files
function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. It also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example: `'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)`

`Modifiable` — Option to create a modifiable protected model
false (default) | true

Option to create a modifiable protected model, specified as a Boolean value. To use this option:

- Add a password for modification using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. If a password has not been added at the time that you create the modifiable protected model, you are prompted to create one.
- Modify the options of your protected model by first providing the modification password using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. Then use the `Simulink.ModelReference.modifyProtectedModel` function to make your option changes.

Example: `'Modifiable',true`

`Callbacks` — Option to specify protected model callbacks
cell array

Option to specify callbacks for a protected model, specified as a cell array of `Simulink.ProtectedModel.Callback` objects.

Example: `'Callbacks',{pmcallback_sim, pmcallback_cg}`

Output Arguments

`harnessHandle` — Handle of the harness model
double

Handle of the harness model, returned as a double or `0`, depending on the value of `Harness`.

If `Harness` is `true`, the value is the handle of the harness model; otherwise, the value is `0`.

`neededVars` — Names of base workspace variables
cell array

Names of base workspace variables used by the model being protected, returned as a cell array.

The cell array can also include variables that the protected model does not use.

Alternatives

"Protect Models to Conceal Contents" (Simulink Coder)

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords` | `Simulink.ModelReference.ProtectedModel.clearPasswordsForModel` | `Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForModify` | `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` |

`Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.modifyProtectedModel`

Topics

“Protect Models to Conceal Contents” (Simulink Coder)
“Protected Models for Model Reference” (Simulink)
“Test Protected Models” (Simulink Coder)
“Package and Share Protected Models” (Simulink Coder)
“Specify Custom Obfuscators for Protected Models” (Simulink Coder)
“Configure and Run SIL Simulation”
“Define Callbacks for Protected Models” (Simulink Coder)
“Reference Protected Models from Third Parties” (Simulink)
“Code Interfaces for SIL and PIL”

Introduced in R2012b

Simulink.ModelReference.ProtectedModel.clearPasswords

Clear cached passwords for protected models

Syntax

```
Simulink.ModelReference.ProtectedModel.clearPasswords()
```

Description

`Simulink.ModelReference.ProtectedModel.clearPasswords()` clears protected model passwords that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear cached passwords for protected models

After using protected models, clear passwords cached for the models during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswords()
```

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel`

Topics

“Protect Models to Conceal Contents” (Simulink Coder)

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.-clearPasswordsForModel

Clear cached passwords for a protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)
```

Description

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)` clears protected model passwords for `model` that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear cached passwords for a protected model

After using a protected model, clear passwords cached for the model during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)
```

Input Arguments

`model` — Protected model name
string or character vector

Model name specified as a string or character vector

Example: '`rtwdemo_counter`'

Data Types: `char`

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords`

Topics

“Protect Models to Conceal Contents” (Simulink Coder)

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.HookInfo class

Package: Simulink.ModelReference.ProtectedModel

Represent files and exported symbols generated by creation of protected model

Description

Specifies information about files and symbols generated when creating a protected model. The creator of a protected model can use this information for postprocessing of the generated files prior to packaging. Information includes:

- List of source code files (`*.c`, `*.h`, `*.cpp`,`*.hpp`).
- List of other related files (`*.mat`, `*.rsp`, `*.prj`, etc.).
- List of exported symbols that you must not modify.

Construction

To access the properties of this class, use the '`CustomPostProcessingHook`' option of the `Simulink.ModelReference.protect` function. The value for the option is a handle to a postprocessing function accepting a `Simulink.ModelReference.ProtectedModel.HookInfo` object as input.

Properties

`ExportedSymbols` — Exported Symbols

cell array of character vectors

A list of exported symbols generated by protected model that you must not modify. Default value is empty.

For a protected model with a top model interface, the `HookInfo` object cannot provide information on exported symbols.

`NonSourceFiles` — Non source code files
cell array of character vectors

A list of non-source files generated by protected model creation. Examples are `*.mat`, `*.rsp`, and `*.prj`. Default value is empty.

`SourceFiles` — Source code files
cell array of character vectors

A list of source code files generated by protected model creation. Examples are `*.c`, `*.h`, `*.cpp`, and `*.hpp`. Default value is empty.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

See Also

`Simulink.ModelReference.protect`

Topics

“Specify Custom Obfuscators for Protected Models” (Simulink Coder)

Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration

Add or provide encryption password for code generation from protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(
model,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(model,password)` adds an encryption password for code generation if you create a protected model. If you use a protected model, the function provides the required password to generate code from the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for code generation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode','Code Generation','Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for code generation.

Generate Code from an Encrypted Protected Model

Use a protected model with encryption for code generation.

Provide the encryption password required for code generation from the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you can generate code from the protected model.

Input Arguments

`model` — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

`password` — Password for protected model code generation

string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for code generation, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForModify

Add or provide password for modifying protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(model,
password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForModify(model, password)` adds a password for a modifiable protected model. After the password has been created, the function provides the password for modifying the protected model.

Examples

Add Functionality to Protected Model

Create a modifiable protected model with support for code generation, then modify it to add Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter', 'password');
```

Add support for Web view to the protected model that you created. Optionally, if you want to add support for HDL code generation, set 'hdl' to true.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Webview',true,...
'Report',true);
```

Input Arguments

model — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model to be modified.

password — Password to modify protected model

string or character vector

Password, specified as a string or character vector. The password is required for modification of the protected model.

See Also

Simulink.ModelReference.modifyProtectedModel | Simulink.ModelReference.protect

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

Add or provide encryption password for simulation of protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(
model,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model,password)` adds an encryption password for simulation if you create a protected model. If you use a protected model, the function provides the required password to simulate the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for simulation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for simulation.

Simulate an Encrypted Protected Model

Use a protected model with encryption for simulation.

Provide the encryption password required for simulation of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you can simulate the protected model.

Input Arguments

`model` — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

`password` — Password for protected model simulation

string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for simulation, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForView

Add or provide encryption password for read-only view of protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(model,
password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForView(model, password)` adds an encryption password for read-only view if you create a protected model. If you use a protected model, the function provides the required password for a read-only view of the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for read-only view.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Webview',true,'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for read-only view.

View an Encrypted Protected Model

Use a protected model with encryption for read-only view.

Provide the encryption password required for the read-only view of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you have access to the read-only view of the protected model.

Input Arguments

`model` — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

`password` — Password for read-only view of protected model

string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for read-only view, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ProtectedModel.addTarget

Add code generation support for current target to protected model

Syntax

```
Simulink.ProtectedModel.addTarget(model)
```

Description

`Simulink.ProtectedModel.addTarget(model)` adds code generation support for the current `model` target to a protected model of the same name. Each target that the protected model supports is identified by the root of the **Code Generation > System Target file** (`SystemTargetFile`) parameter. For example, if the **System Target file** is `ert.tlc`, the target identifier is `ert`.

To add the current target:

- The model and the protected model of the same name must be on the MATLAB path.
- The protected model must have the `Modifiable` option enabled and have a password for modification.
- The target must be unique in the protected model.

If you add a target to a protected model that did not previously support code generation, the software switches the protected model `Mode` to `CodeGeneration` and `ObfuscateCode` to `true`.

Examples

Add a Target to a Protected Model

Add the currently configured model target to the protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get a list of targets that the protected model supports.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the unprotected model to support a new target.

```
 set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
 save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
 Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`model` — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` |

`Simulink.ProtectedModel.getSupportedTargets` |
`Simulink.ProtectedModel.removeTarget` |
`Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create Protected Models with Multiple Targets” (Simulink Coder)

Introduced in R2015a

Simulink.ProtectedModel.Callback class

Package: Simulink.ProtectedModel

Represents callback code that executes in response to protected model events

Description

For a protected model functionality, the `Simulink.ProtectedModel.Callback` object specifies code to execute in response to an event. The callback code can be a character vector of MATLAB commands or a MATLAB script. The object includes:

- The code to execute for the callback.
- The event that triggers the callback.
- The protected model functionality that the event applies to.
- The option to override the protected model build.

When you create a protected model, to specify callbacks, call the Simulink.ModelReference.protect function with the `'Callbacks'` option. The value of this option is a cell array of `Simulink.ProtectedModel.Callback` objects.

Construction

`pmCallback = Simulink.ProtectedModel.Callback(event,functionality, callbackText)` creates a callback object for a specific protected model functionality and event. The `callbackText` specifies MATLAB commands to execute for the callback.

`pmCallback = Simulink.ProtectedModel.Callback(event,functionality, callbackFile)` creates a callback object for a specific protected model functionality and event. The `callbackFile` specifies a MATLAB script to execute for the callback. The script must be on the MATLAB path.

Input Arguments

`event` — Event that triggers callback
`'PreAccess'` | `'Build'`

Callback trigger event. Specify one of the following values:

- `'PreAccess'`: Callback code is executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code is executed before build. Valid only for `'CODEGEN'` functionality.

`functionality` — Protected model functionality
`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Specify one of the following values:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If you do not specify a functionality, the default behavior is `'AUTO'`.

`callbackText` — Callback code to execute
string or character vector

MATLAB commands to execute in response to an event, specified as a string or character vector.

`callbackFile` — Callback script to execute
string or character vector

MATLAB script to execute in response to an event, specified as a string or character vector. Script must be on the MATLAB path.

Properties

`AppliesTo` — Protected model functionality
`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Value is one of the following:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If you do not specify a functionality, the default behavior is `'AUTO'`.

`CallbackFileName` — Callback script to execute

string or character vector

MATLAB script to execute in response to an event, specified as a string or character vector. Script must be on the MATLAB path.

Example: `'pmCallback.m'`

`CallbackText` — Callback code to execute

string or character vector

MATLAB commands to execute in response to an event, specified as a string or character vector.

Example: `'A = [15 150];disp(A)'`

`Event` — Event that triggers callback

`'PreAccess'` | `'Build'`

Callback trigger event. Value is one of the following:

- `'PreAccess'`: Callback code is executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code is executed before build. Valid only for `'CODEGEN'` functionality.

`OverrideBuild` — Option to override protected model build

`false` (default) | `true`

Option to override the protected model build process, specified as a Boolean value. Applies only to a callback object that you define for a `'Build'` event for `'CODEGEN'` functionality. You set this option using the `setOverrideBuild` method.

Methods

setOverrideBuild	Specify option to override protected model build

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create Protected Model Using a Callback

Create a callback object with a character vector of MATLAB commands for the callback code. Specify the object when you create a protected model.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess',...
'SIM','disp(''Hello world!'')')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Callbacks',{pmCallback})
sim('sldemo_mdlref_basic')
```

For each instance of the protected model reference in the top model, the output is listed.

```
Hello world!
Hello world!
Hello world!
```

Create Protected Model With a Callback Script

Create a callback object with a MATLAB script for the callback code. Specify the object when you create a protected model.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
rtwbuild('sldemo_mdlref_basic')
```

Before the protected model build process begins, code in `pm_callback.m` executes.

See Also

`Simulink.ModelReference.protect` |
`Simulink.ProtectedModel.getCallbackInfo`

Topics

“Define Callbacks for Protected Models” (Simulink Coder)
“Protect Models to Conceal Contents” (Simulink Coder)
“Code Generation Requirements and Limitations” (Simulink Coder)

Introduced in R2016a

setOverrideBuild

Class: Simulink.ProtectedModel.Callback
Package: Simulink.ProtectedModel

Specify option to override protected model build

Syntax

```
setOverrideBuild(override)
```

Description

`setOverrideBuild(override)` specifies whether a `Simulink.ProtectedModel.Callback` object can override the build process. This method is valid only for callbacks that execute in response to a `'Build'` event for `'CODEGEN'` functionality.

Input Arguments

`override` — Option to override protected model build process

false (default) | true

Option to override the protected model build process, specified as a Boolean value. This option applies only to a callback object defined for a `'Build'` event for `'CODEGEN'` functionality.

Example: `pmcallback.setOverrideBuild(true)`

Examples

Create Code Generation Callback to Override Build Process

Create a callback object with a character vector of MATLAB commands for the callback code. Specify that the callback override the build process.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','disp(''Hello world!'')')
pmCallback.setOverrideBuild(true);
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
rtwbuild('sldemo_mdlref_basic')
```

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.Callback`

Topics

“Define Callbacks for Protected Models” (Simulink Coder)
“Protect Models to Conceal Contents” (Simulink Coder)
“Code Generation Requirements and Limitations” (Simulink Coder)

Introduced in R2016a

Simulink.ProtectedModel.CallbackInfo class

Package: Simulink.ProtectedModel

Protected model information for use in callbacks

Description

A `Simulink.ProtectedModel.CallbackInfo` object contains information about a protected model that you can use in the code executed for a callback. The object provides:

- Model name.
- List of models and submodels in the protected model container.
- Callback event.
- Callback functionality.
- Code interface.
- Current target. This information is available only for code generation callbacks.

Construction

`cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event,functionality)` creates a `Simulink.ProtectedModel.CallbackInfo` object.

Properties

`CodeInterface` — Code interface generated by protected model

`'Top model'` | `'Model reference'`

Code interface that the protected model generates.

`Event` — Event that triggered callback

`'PreAccess'` | `'Build'`

Callback trigger event. Value is one of the following:

- `'PreAccess'`: Callback code executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code executed before build. Valid only for `'CODEGEN'` functionality.

`Functionality` — Protected model functionality
`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Value is one of the following:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If the value of `functionality` is blank, the default behavior is `'AUTO'`.

`ModelName` — Protected model name
character vector

Protected model name, specified as a character vector.

`SubModels` — Models and submodels in the protected model container
cell array of character vectors

Names of all models and submodels in the protected model container, specified as a cell array of character vectors.

`Target` — Current target
character vector

Current target identifier for the protected model, specified as a character vector. This property is available only for code generation callbacks.

Methods

getBuildInfoForModel	Get build information object for specified model

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Use Protected Model Information in Simulation Callback

Create a protected model callback that uses information from the `Simulink.ProtectedModel.Callback` object.

First, on the MATLAB path, create a callback script , `pm_callback.m`, containing:

```
s1 = 'Simulating protected model: ';
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','PreAccess','SIM');
disp([s1 cbinfobj.ModelName])
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess'...
,'SIM', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Callbacks',{pmCallback})
```

Simulate the protected model. For each instance of the protected model reference in the top model, the output from the callback is listed.

```
sim('sldemo_mdlref_basic')
```

```
Simulating protected model: sldemo_mdlref_counter
Simulating protected model: sldemo_mdlref_counter
Simulating protected model: sldemo_mdlref_counter
```

See Also

`Simulink.ModelReference.protect` |
`Simulink.ProtectedModel.getCallbackInfo`

Topics

“Define Callbacks for Protected Models” (Simulink Coder)
“Protect Models to Conceal Contents” (Simulink Coder)
“Code Generation Requirements and Limitations” (Simulink Coder)

Introduced in R2016a

Simulink.ProtectedModel.getCallbackInfo

Get `Simulink.ProtectedModel.CallbackInfo` object for use by callbacks

Syntax

```
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event,
functionality)
```

Description

`cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event, functionality)` returns a `Simulink.ProtectedModel.CallbackInfo` object that provides information for protected model callbacks. The object contains information about the protected model, including:

- Model name.
- List of models and submodels in the protected model container.
- Callback event.
- Callback functionality.
- Code interface.
- Current target. This information is available only for code generation callbacks.

Examples

Use Protected Model Information in Code Generation Callback

On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
s1 = 'Code interface is: ';
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
disp([s1 cbinfobj.CodeInterface]);
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

Build the protected model. Before the start of the protected model build process, the code interface is displayed.

```
rtwbuild('sldemo_mdlref_basic')
```

Input Arguments

`modelName` — Protected model name

string or character vector

Protected model name, specified as a string or character vector.

`event` — Event that triggered callback

`'PreAccess'` | `'Build'`

Callback trigger event. Value is one of the following:

- `'PreAccess'`: Callback code executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code executed before build. Valid only for `'CODEGEN'` functionality.

`functionality` — Protected model functionality

`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Value is one of the following:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If the value of `functionality` is blank, the default behavior is `'AUTO'`.

Output Arguments

`cbinfobj` — Callback information object
`Simulink.ProtectedModel.CallbackInfo`

Callback information, specified as a `Simulink.ProtectedModel.CallbackInfo` object.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

Topics

"Define Callbacks for Protected Models" (Simulink Coder)
"Protect Models to Conceal Contents" (Simulink Coder)
"Code Generation Requirements and Limitations" (Simulink Coder)

Introduced in R2016a

getBuildInfoForModel

Class: Simulink.ProtectedModel.CallbackInfo
Package: Simulink.ProtectedModel

Get build information object for specified model

Syntax

```
bldobj = getBuildInfoForModel(model)
```

Description

`bldobj = getBuildInfoForModel(model)` returns a handle to an `RTW.BuildInfo` object. This object specifies the build toolchain and arguments. The `model` name must be in the list of model names in the `SubModels` property of the `Simulink.ProtectedModel.CallbackInfo` object. You can call this method only for code generation callbacks in response to a `'Build'` event.

Input Arguments

`model` — Model name

string or character vector

Model name, specified as a string or character vector. The `model` name must be in the list of model names in the `SubModels` property of the `Simulink.ProtectedModel.CallbackInfo` object. You can call this method only for code generation callbacks in response to a `'Build'` event.

Output Arguments

`bldobj` — Object for build toolchain and arguments

`RTW.BuildInfo`

Build toolchain and arguments, specified as a `RTW.BuildInfo` object. If you do not call the method for a code generation callback and `'Build'` event, the return value is an empty array.

Examples

Get Build Information from a Code Generation Callback

On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
bldinfo = cbinfobj.getBuildInfoForModel(cbinfobj.ModelName);
buildargs = getBuildArgs(bldinfo)
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

Build the protected model. Before the start of the protected model build, the build arguments are displayed.

```
rtwbuild('sldemo_mdlref_basic')
```

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

Topics

“Define Callbacks for Protected Models” (Simulink Coder)
“Protect Models to Conceal Contents” (Simulink Coder)
“Code Generation Requirements and Limitations” (Simulink Coder)

Introduced in R2016a

Simulink.ProtectedModel.getConfigSet

Get configuration set for current protected model target or for specified target

Syntax

```
configSet = Simulink.ProtectedModel.getConfigSet(protectedModel)
configSet = Simulink.ProtectedModel.getConfigSet(protectedModel,
targetID)
```

Description

`configSet = Simulink.ProtectedModel.getConfigSet(protectedModel)` returns the configuration set object for the current, protected model target.

`configSet = Simulink.ProtectedModel.getConfigSet(protectedModel, targetID)` returns the configuration set object for a specified target that the protected model supports.

Examples

Get Configuration Set for Current Target

Get the configuration set for the currently configured, protected model target.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get the configuration set for the currently configured target.

```
cs = Simulink.ProtectedModel.getConfigSet('mdlref_counter')
```

Get Configuration Set for Specified Target

Get the configuration set for a specified target that the protected model supports.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
 set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
 save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
 Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Get the configuration set for the added target.

```
cs = Simulink.ProtectedModel.getConfigSet('mdlref_counter','ert')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target identifier
string or character vector

Identifier for selected target, specified as a string or character vector. The target identifier is the root of the **Code Generation > System Target file** (`SystemTargetFile`) parameter. For example, if the **System Target file** is `ert.tlc`, the target identifier is `ert`.

Output Arguments

`configSet` — Configuration object
`Simulink.ConfigSet`

Configuration set, specified as a `Simulink.ConfigSet` object

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget` | `Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create Protected Models with Multiple Targets” (Simulink Coder)
“Reference Protected Models from Third Parties” (Simulink)

Introduced in R2015a

Simulink.ProtectedModel.getCurrentTarget

Get current protected model target

Syntax

```
currentTarget = Simulink.ProtectedModel.getCurrentTarget(
protectedModel)
```

Description

`currentTarget = Simulink.ProtectedModel.getCurrentTarget(protectedModel)` returns the target identifier for the target that is currently configured for the protected model. At the start of a MATLAB session, the default current target is the last target added to the protected model. Otherwise, the current target is the last target that you used. You can change the current target using the `Simulink.ProtectedModel.setCurrentTarget` function.

When building the model, the software changes the target to match the parent if the currently selected target does not match the target of the parent model.

Examples

Get Currently Configured Target for Protected Model

Add a target to a protected model, and then get the currently configured target for the protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Get the currently configured target for the protected model.

```
ct = Simulink.ProtectedModel.getCurrentTarget('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

Output Arguments

`currentTarget` — Current target
character vector

Current target for protected model, specified as a character vector.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget` | `Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create Protected Models with Multiple Targets” (Simulink Coder)
“Reference Protected Models from Third Parties” (Simulink)

Introduced in R2015a

Simulink.ProtectedModel.getSupportedTargets

Get list of targets that protected model supports

Syntax

```
supportedTargets = Simulink.ProtectedModel.getSupportedTargets(
protectedModel)
```

Description

`supportedTargets = Simulink.ProtectedModel.getSupportedTargets(protectedModel)` returns a list of target identifiers for the code generation targets supported by the specified protected model. The target identifier `sim` represents simulation support.

Examples

Get List of Supported Targets for a Protected Model

Add a target to a protected model, and then get a list of supported targets to verify the addition of the new target.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

Output Arguments

`supportedTargets` — List of target identifiers
cell array of character vectors

List of target identifiers for the targets that the protected model supports, specified as a cell array of character vectors.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` |

`Simulink.ProtectedModel.removeTarget` |
`Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create Protected Models with Multiple Targets” (Simulink Coder)
“Reference Protected Models from Third Parties” (Simulink)

Introduced in R2015a

Simulink.ProtectedModel.open

Open protected model

Syntax

```
Simulink.ProtectedModel.open(model)
Simulink.ProtectedModel.open(model,type)
```

Description

`Simulink.ProtectedModel.open(model)` opens a protected model. If you do not specify how to view the protected model, the software first tries to open the Web view. If the Web view is not enabled for the protected model, the software then tries to open the report. If you did not create a report, the software reports an error.

`Simulink.ProtectedModel.open(model,type)` opens a protected model using the specified viewing method. If you specify 'webview', the software opens the Web view for the protected model. If you specify 'report', the software opens the protected model report. If the method that you specify is not enabled, the software reports an error. The protected model is not opened.

Examples

Open a Protected Model

Open a protected model without a specified method.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model enabling support for code generation and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Report',true);
```

Open the protected model without specifying how to view it.

```
Simulink.ProtectedModel.open('mdlref_counter')
```

The protected model does not have Web view enabled, so the protected model report is opened.

Open a Protected Model Web View

Open a protected model, specifying the Web view.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model with support for code generation, Web view, and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Webview',true,'Report',true);
```

Open the protected model and specify that you want to see the Web view.

```
Simulink.ProtectedModel.open('mdlref_counter','webview')
```

The protected model Web view is opened.

Input Arguments

`model` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`type` — Open method
'webview' | 'report'

Method for viewing the protected model. If you specify ‘webview’, the software opens the Web view for the protected model. If you specify ‘report’, the software opens the protected model report.

See Also

`Simulink.ModelReference.protect`

Introduced in R2015a

Simulink.ProtectedModel.removeTarget

Remove support for specified target from protected model

Syntax

```
Simulink.ProtectedModel.removeTarget(protectedModel,targetID)
```

Description

`Simulink.ProtectedModel.removeTarget(protectedModel,targetID)` removes code generation support for the specified target from a protected model. You must provide the modification password to make this update. Removing a target does not require access to the unprotected model.

Note You cannot remove the `sim` target. If you do not want the protected model to support simulation, use the `Simulink.ModelReference.modifyProtectedModel` function to change the protected model mode to `ViewOnly`.

Examples

Remove Target Support from a Protected Model

Remove a supported target from a protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Remove support for the `ert` target from the protected model. You are prompted for the modification password.

```
Simulink.ProtectedModel.removeTarget('mdlref_counter','ert');
```

Verify that support for the `ert` target has been removed from the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name

string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target to be removed

string or character vector

Identifier for target to be removed, specified as a string or character vector.

See Also

`Simulink.ModelReference.modifyProtectedModel` | `Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.setCurrentTarget`

Topics

"Create Protected Models with Multiple Targets" (Simulink Coder)

Introduced in R2015a

Simulink.ProtectedModel.setCurrentTarget

Configure protected model to use specified target

Syntax

```
Simulink.ProtectedModel.setCurrentTarget(protectedModel, targetID)
```

Description

`Simulink.ProtectedModel.setCurrentTarget(protectedModel, targetID)` configures the protected model to use the target that the target identifier specifies.

Note If you include the protected model in a model reference hierarchy, the software tries to change the current target to match the target of the parent model. If the software cannot match the target of the parent, it reports an error.

Examples

Set Current Target for Protected Model

After you get a list of supported targets, set the current target for a protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get a list of targets that the protected model supports.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the protected model to use the new target.

```
Simulink.ProtectedModel.setCurrentTarget('mdlref_counter','ert');
```

Verify that the current target is the new target.

```
ct = Simulink.ProtectedModel.getCurrentTarget('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target identifier
string or character vector

Identifier for selected target, specified as a string or character vector.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget`

Topics

“Create Protected Models with Multiple Targets” (Simulink Coder)
“Reference Protected Models from Third Parties” (Simulink)

Introduced in R2015a

slConfigUIGetVal

Return current value for custom target configuration option

Syntax

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

Input 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 another 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 another purpose.

`'OptionName'`

Quoted name of the TLC variable defined for a custom target configuration option.

Output Arguments

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 you:

- Load the model.

- Update configuration settings in the Configuration Parameters dialog box.
- Build the model.

You use `slConfigUIGetVal` to read the current value of a specified target option.

Examples

In the following example, the `slConfigUIGetVal` function returns the value of the **Configuration Parameters > Code Generation > Interface > Advanced parameters > Terminate function required** option.

```
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`

Topics

“Define and Display Custom Target Options” (Simulink Coder)
“Custom Target Optional Features” (Simulink Coder)

Introduced in R2006b

slConfigUISetEnabled

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 another 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 another 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.

If you use this function to disable a parameter that is represented in the Configuration Parameters dialog box, the parameter appears greyed out in the dialog context.

Examples

In the following example, the `slConfigUISetEnabled` function disables the **Configuration Parameters > Code Generation > Interface > Advanced parameters > Terminate function required** option.

```
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`

Topics

“Define and Display Custom Target Options” (Simulink Coder)
“Custom Target Optional Features” (Simulink Coder)

Introduced in R2006b

slConfigUISetVal

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 another 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 another 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.

Examples

In the following example, the `slConfigUISetVal` function sets the value `'off'` for the **Configuration Parameters > Code Generation > Interface > Advanced parameters > Terminate function required** option.

```
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`

Topics

“Define and Display Custom Target Options” (Simulink Coder)
“Custom Target Optional Features” (Simulink Coder)

Introduced in R2006b

switchTarget

Select target for model configuration set

Syntax

```
switchTarget(myConfigObj,systemTargetFile,[])
switchTarget(myConfigObj,systemTargetFile,targetOptions)
```

Description

`switchTarget(myConfigObj,systemTargetFile,[])` changes the selected system target file for the active configuration set.

`switchTarget(myConfigObj,systemTargetFile,targetOptions)` sets the configuration parameters specified by `targetOptions`.

Examples

Get ConfigSet, Default Options, and Switch Target

This example shows how to get the active configuration set for `model`, and change the system target file for the configuration set.

```
% Get configuration set for model
myConfigObj = getActiveConfigSet(model);
% Switch system target file
switchTarget(myConfigObj,'ert.tlc',[]);
```

Get ConfigSet, Set Options, Switch Target

This example shows how to get the active configuration set for the current model (`gcs`), set various `targetOptions`, then change the system target file selection.

```
% Get configuration set for current model
myConfigObj=getActiveConfigSet(gcs);

% Specify target options
targetOptions.TLCOptions = '-aVarName=1';
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = 'my target';
targetOptions.TemplateMakefile = 'grt_default_tmf';

% Define a system target file
targetSystemFile='grt.tlc';

% Switch system target file
switchTarget(myConfigObj,targetSystemFile,targetOptions);
```

Use `targetOptions` to verify values (optional).

```
% Verify values (optional)
targetOptions
```

```
          TLCOptions: '-aVarName=1'
         MakeCommand: 'make_rtw'
         Description: 'my target'
    TemplateMakefile: 'grt_default_tmf'
```

Get ConfigSet, Set Options for MSVC Solution Build, Switch Target to MSVC ERT

This example shows how to get the active configuration set for `model`, then change the system target file to the ERT Create Visual C/C++ Solution File for Embedded Coder.

```
model='rtwdemo_rtwintro';
open_system(model);

% Get configuration set for model
myConfigObj = getActiveConfigSet(model);

% Specify target options for MSVC build
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = ...
     'Create Visual C/C++ Solution File for Embedded Coder';
targetOptions.TemplateMakefile = 'RTW.MSVCBuild';
```

```
% Switch system target file
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Get ConfigSet, Set Options for Toolchain Build, and Switch Target

Use options to select default ERT target file, instead of `set_param(model,'SystemTargetFile','ert.tlc')`.

```
% use switchTarget to select toolchain build of defaul ERT target
model='rtwdemo_rtwintro';
open_system(model);

% Get configuration set for model
myConfigObj = getActiveConfigSet(model);

% Specify target options for toolchain build approach
targetOptions.MakeCommand = '';
targetOptions.Description = 'Embedded Coder';
targetOptions.TemplateMakefile = '';

% Switch system target file
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Input Arguments

`myConfigObj` — Configuration set object

`object`

A configuration set object of `Simulink.ConfigSet` or configuration reference object of `Simulink.ConfigSetRef`. Call `getActiveConfigSet` to get the configuration set object.

Example: `myConfigObj = getActiveConfigSet(model);`

`systemTargetFile` — Name of system target file

character vector

Specify the name of the system target file (such as `ert.tlc` for Embedded Coder or `grt.tlc` for Simulink Coder) as the name appears in the **System Target File Browser**.

Example: `systemTargetFile = 'ert.tlc';`

`targetOptions` — Structure with field values that provide configuration parameter options
struct

Structure with fields that define a code generation target options. You can choose to modify certain configuration parameters by filling in values in a structure field. If you do not want to use options, specify an empty structure (`[]`).

Field Values in targetOptions

Specify the structure field values of the `targetOptions`. If you choose not to specify options, use an empty structure (`[]`).

Example: `targetOptions = [];`

`TemplateMakefile` — Character vector specifying file name of template makefile
character vector

Example: `targetOptions.TemplateMakefile = 'RTW.MSVCBuild';`

`TLCOptions` — Character vector specifying TLC argument
character vector

Example: `targetOptions.TLCOptions = '-aVarName=1';`

`MakeCommand` — Character vector specifying make command MATLAB language file
character vector

Example: `targetOptions.MakeCommand = 'make_rtw';`

`Description` — Character vector specifying description of the system target file
character vector

Example: `targetOptions.Description = 'Create Visual C/C++ Solution File for Embedded Coder';`

See Also

`Simulink.ConfigSet` | `Simulink.ConfigSetRef` | `getActiveConfigSet`

Topics

“Select a System Target File Programmatically” (Simulink Coder)

“Configure a System Target File” (Simulink Coder)
“Set Target Language Compiler Options” (Simulink Coder)

Introduced in R2009b

target Package

Register new target hardware

Description

Manage target hardware information

Classes

target.LanguageImplementation	Provide C and C++ compiler implementation details
target.Processor	Provide target processor information

Functions

target.add	Add target feature object to MATLAB memory
target.create	Create target feature object
target.get	Retrieve target feature object from MATLAB memory
target.remove	Remove target feature object from MATLAB memory

See Also

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.add

Package: target

Add target feature object to MATLAB memory

Syntax

```
target.add(targetFeatureObject)
target.add(targetFeatureObject, 'UserInstall', dataPersistence)
```

Description

`target.add(targetFeatureObject)` adds the specified target feature object to MATLAB memory. By default, the target data is available only for the current MATLAB session.

`target.add(targetFeatureObject, 'UserInstall', dataPersistence)` controls persistence of target data over MATLAB sessions.

Examples

Create New Hardware Implementation

For workflow examples that use this function, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

Input Arguments

`targetFeatureObject` — Target feature object
object

Specify the target feature object that you want to add to MATLAB memory.

Example: `target.add(myTargetFeatureObject);`

`dataPersistence` — Target data persistence
false (default) | true

Control persistence of target data in MATLAB memory:

- `true` –- Target data persists over multiple MATLAB sessions.
- `false` –- Target data is available only for the current MATLAB session.

Example: `target.add(myTargetFeatureObject, 'UserInstall', true);`

Data Types: `logical`

See Also

`target.create` | `target.get` | `target.remove`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.create

Package: target

Create target feature object

Syntax

```
targetFeatureObject = target.create(targetFeatureClass)
targetFeatureObject = target.create(targetFeatureClass,Name,Value)
```

Description

`targetFeatureObject = target.create(targetFeatureClass)` creates and returns an object of the specified class.

`targetFeatureObject = target.create(targetFeatureClass,Name,Value)` configures the object using one or more name-value pair arguments.

Examples

Create New Hardware Implementation

For workflow examples that use this function, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

Input Arguments

`targetFeatureClass` — Target feature class
character vector | string

Specify class of object:

- `'Processor'`-- Create `target.Processor` object.
- `'LanguageImplementation'`-- Create `target.LanguageImplementation` object.
- `'Alias'`-- Create `target.Alias` object.

Example: `'Processor'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `myLangImp = target.create('LanguageImplementation', 'Name', 'myLanguageImplementation', 'Copy', 'ARM Compatible-ARM Cortex');`

Example: `myProc = target.create('Processor', 'Name', 'myProcessor', 'Manufacturer', 'myProcessorManufacturer');`

`Copy` — Copy existing target feature object
character vector | string

Create a target feature object by copying values from an existing target feature object.

***`propertyName`* — Property name**
character vector | string

Create the target feature object with properties that are set to values that you specify.

Output Arguments

`targetFeatureObject` — Target feature object
object

The returned object is a:

- `target.Processor` object if `targetFeatureClass` is `'Processor'`
- `target.LanguageImplementation` object if `targetFeatureClass` is `'LanguageImplementation'`
- `target.Alias` object if `targetFeatureClass` is `'Alias'`

See Also

`target.add` | `target.get` | `target.remove`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.get

Package: target

Retrieve target feature object from MATLAB memory

Syntax

```
targetFeatureObject = target.get(targetFeatureClass,
targetFeatureObjectId)
```

Description

`targetFeatureObject = target.get(targetFeatureClass, targetFeatureObjectId)` retrieves a target feature object from MATLAB memory.

Examples

Remove Target Feature Object

This example shows how you can remove a `target.LanguageImplementation` object associated with an object identifier, *`myLanguageImplementationID`*.

Retrieve the object from MATLAB memory.

```
objectToRemove = target.get('LanguageImplementation', myLanguageImplementationID);
```

Remove the object.

```
target.remove(objectToRemove);
```

Input Arguments

`targetFeatureClass` — Target feature class
character vector | string

Specify the class of the object that you want to retrieve. For example, to retrieve:

- A `target.Processor` object, specify `'Processor'` .
- A `target.LanguageImplementation` object, specify `'LanguageImplementation'`.

`targetFeatureObjectId` — Target feature object identifier
character vector | string

Specify the unique identifier of the object that you want to retrieve, that is, the `Id` property value of the object.

Output Arguments

`targetFeatureObject` — Target feature object
object

Retrieved target feature object. For example:

- If `targetFeatureClass` is `'Processor'`, the returned object is a `target.Processor` object.
- If `targetFeatureClass` is `'LanguageImplementation'`, the returned object is a `target.LanguageImplementation` object.

See Also
`target.add` | `target.create` | `target.remove`

Topics
“Register New Hardware Devices”

Introduced in R2019a

target.LanguageImplementation class

Package: target

Provide C and C++ compiler implementation details

Description

Use the `target.LanguageImplementation` class to provide implementation details about the C and C++ compiler for your target hardware. For example, byte ordering.

To create a `target.LanguageImplementation` object, use the `target.create` function.

Properties

`AtomicFloatSize` — Largest atomic float size

integer

Size in bits of the largest floating-point data type that you can atomically load and store on the hardware

Attributes:

GetAccess

public

SetAccess

public

Data Types: `int32`

`AtomicIntegerSize` — Largest atomic integer size

integer

Size in bits of the largest integer that you can atomically load and store on the hardware

Attributes:

GetAccess
public
SetAccess
public

Data Types: `int32`

`Endianess` — Byte ordering

`'Little'` (default) | `'Big'` | `'Unspecified'`

Byte ordering implemented by target hardware.

Attributes:

GetAccess
public
SetAccess
public

`DataTypes` — Data type definitions

object

`target.DataTypes` object that provides C and C++ data type definitions through properties described in this table.

Property Name	Purpose
`Char`	`target.datatype.Char` object for `char`.
`Short`	`target.datatype.Short` object for `short`.
`Int`	`target.datatype.Int` object for `int`.
`Long`	`target.datatype.Long` object for `long`.
`LongLong`	`target.datatype.LongLong` object for `longlong`.
`Half`	`target.datatype.Half` object for a half precision data type that the target uses.
`Float`	`target.datatype.Float` object for `float`.
`Double`	`target.datatype.Double` object for `double`.

Property Name	Purpose
`Pointer`	`target.datattype.Pointer` object for `pointer`.
`SizeT`	`target.datatype.SizeT` object for `size_t`.
`PtrDiffT`	`target.datatype.PtrDiffT` object for `ptrdiff_t`.

Attributes:

GetAccess
: public

SetAccess
: private

`Id` — Object identifier

character vector | string

Value of `Name` property.

Attributes:

GetAccess
: public

SetAccess
: private

`Name` — Name

character vector | string

Name of the target language implementation

Attributes:

GetAccess
: public

SetAccess
: public

`WordSize` — Native word size

integer

Native word size for the target hardware.

Attributes:

GetAccess
: public

SetAccess
: public

Data Types: `int32`

Examples

Create New Hardware Implementation

For examples that use this class, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

See Also

`target.Processor` | `target.create`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.Processor class

Package: target

Provide target processor information

Description

Use the `target.Processor` class to provide information about your target processor. For example, name, manufacturer, and language implementation.

To create a `target.Processor` object, use the `target.create` function.

Properties

`Id` — Object identifier

character vector | string

The object identifier is the hyphenated combination of the `Manufacturer` and `Name` property values. If the `Manufacturer` property is empty, the object identifier is the `Name` property value.

Attributes:

GetAccess

public

SetAccess

private

`LanguageImplementations` — Language implementation

object

Associated `target.LanguageImplementation` object.

Attributes:

GetAccess
public
SetAccess
public

`Name` — Processor name

character vector | string

Name of the target processor.

Example: `'Cortex-A53'`

Attributes:

GetAccess
public
SetAccess
public

`Manufacturer` — Processor manufacturer

character vector | string

Optional description of the target processor manufacturer.

Example: `'ARM Compatible'`

Attributes:

GetAccess
public
SetAccess
public

Examples

Create New Hardware Implementation

For examples that use this class, see:

- “Specify Hardware Implementation for New Device”
- “Create New Hardware Implementation By Modifying Existing Implementation”
- “Create New Hardware Implementation By Reusing Existing Implementation”

See Also

`target.LanguageImplementation` | `target.create`

Topics

“Register New Hardware Devices”

Introduced in R2019a

target.remove

Package: target

Remove target feature object from MATLAB memory

Syntax

```
target.remove(targetFeatureObject)
target.remove(targetFeatureClass, targetFeatureObjectId)
```

Description

`target.remove(targetFeatureObject)` removes the target feature object from MATLAB memory.

`target.remove(targetFeatureClass, targetFeatureObjectId)` removes the target feature object specified by class and identifier.

Examples

Remove Target Feature Object From Memory

You can specify and save a hardware device implementation to MATLAB memory.

```
armv8 = target.create('LanguageImplementation', ...
                      'Name', 'Armv8-A LP64');
a53 = target.create('Processor', 'Name', 'Cortex-A53', ...
                    'Manufacturer', 'ARM Compatible');
a53.LanguageImplementations = armv8;
target.add(a53)
```

When target feature objects are not required, you can use the function to remove the objects from MATLAB memory.

To remove only the `target.Processor` object, run:

```
target.remove(a53)
```

or:

```
target.remove('Processor', 'ARM Compatible-Cortex-A53');
```

Input Arguments

`targetFeatureObject` — Target feature object

object

Specify the target feature object that you want to remove.

`targetFeatureClass` — Target feature class

character vector | string

Specify the class of the target feature object that you want to remove. For example:

- If the class is `target.Processor`, specify `'Processor'`.
- If the class is `target.LanguageImplementation`, specify `'LanguageImplementation'`.

Example: `'Processor'`

`targetFeatureObjectId` — Target feature object identifier

character vector | string

Specify the unique identifier of the object that you want to remove, that is, the `Id` property value of the object.

See Also

`target.add` | `target.create` | `target.get`

Topics

“Register New Hardware Devices”

Introduced in R2019a

tlc

Invoke Target Language Compiler to convert model description file to generated code

Syntax

`tlc [-options] [file]`

Description

`tlc [-options] [file]` invokes the Target Language Compiler (TLC) from the command prompt. The TLC converts the model description file, *`model`*`.rtw` (or similar files), into target-specific code or text. Typically, you do not call this command because the build process automatically invokes the Target Language Compiler when generating code. For more information, see “Target Language Compiler Basics” (Simulink Coder).

Note This command is used only when invoking the TLC separately from the build process. You cannot use this command to initiate code generation for a model.

You can change the default behavior by specifying one or more compilation *options* as described in “Options” on page 2-302

Options

You can specify one or more compilation options with each `tlc` command. Use spaces to separate options and arguments. TLC resolves options from left to right. If you use conflicting options, the right-most option prevails. The `tlc` options are:

- “-r Specify model.rtw file name” on page 2-303
- “-v Specify verbose level” on page 2-303
- “-l Specify path to local include files” on page 2-303
- “-m Specify maximum number of errors” on page 2-303

-r Specify *model*.rtw file name

`-r` *file_name*

Specify the file name that you want to translate.

-v Specify verbose level

`-v` *number*

Specify a `number` indicating the verbose level. If you omit this option, the default value is one.

-l Specify path to local include files

`-l` *path*

Specify a folder `path` to local include files. The TLC searches this path in the order specified.

-m Specify maximum number of errors

`-m` *number*

Specify the maximum `number` of errors reported by the TLC prior to terminating the translation of the `.tlc` file.

If you omit this option, the default value is five.

-O Specify the output file path

`-O` *path*

Specify the folder `path` to place output files.

If you omit this option, TLC places output files in the current folder.

-d[a|c|n|o] Invoke debug mode

`-da` execute any %assert directives

`-dc` invoke the TLC command line debugger

`-dn` produce log files, which indicate those lines hit and those lines missed during compilation.

`-do` disable debugging behavior

-a Specify parameters

`-a` *identifier* = *expression*

Specify parameters to change the behavior of your TLC program. For example, this option is used by the code generator to set inlining of parameters or file size limits.

-p Print progress

`-p` *number*

Print a '.' indicating progress for every `number` of TLC primitive operations executed.

-lint Performance checks and runtime statistics

`-lint`

Perform simple performance checks and collect runtime statistics.

-xO Parse only

`-x0`

Parse only a TLC file; do not execute it.

Introduced in R2009a

updateFilePathsAndExtensions

Update files in model build information with missing paths and file extensions

Syntax

```
updateFilePathsAndExtensions(buildinfo,extensions)
```

Description

`updateFilePathsAndExtensions(buildinfo,extensions)` specifies the file name extensions (file types) to include in search and update processing.

Using paths from the build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information require an updated path or file extension. Use this function to:

- Maintain build information for a toolchain that requires the use of file extensions.
- Update multiple customized instances of build information for a given model.

If you use `updateFilePathsAndExtensions`, you call it after you add files to the build information. This approach minimizes the potential performance impact of the required disk I/O.

Examples

Update File Paths and Extensions in Build Information

In your working folder, create the folder path `etcproj/etc` , add files `etc.c`, `test1.c`, and `test2.c` to the folder `etc`. For this example, the working folder is `w:\work \BuildInfo`. From the working folder, update build information `myModelBuildInfo` with missing paths or file extensions.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,fullfile(pwd, ...
```

```
   'etcproj','/etc'),'test');
addSourceFiles(myModelBuildInfo,{'etc' 'test1' ...
   'test2'},'','test');
before = getSourceFiles(myModelBuildInfo,true,true);


>> before

before =

    '\etc'    '\test1'    '\test2'

updateFilePathsAndExtensions(myModelBuildInfo);
after = getSourceFiles(myModelBuildInfo,true,true);


>> after{:}

ans =

    'w:\work\BuildInfo\etcproj\etc\etc.c'


ans =

    'w:\work\BuildInfo\etcproj\etc\test1.c'


ans =

    'w:\work\BuildInfo\etcproj\etc\test2.c'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`extensions` — File name extensions to include in search and update processing `'.c'` (default) | cell array of character vectors | string

The *`extensions`* argument specifies the file name extensions (file types) to include in search and update processing. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For

example, if you specify `{'.c' '.cpp'}` and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` is updated to `myfile.c`.

Example: `'.c' '.cpp'`

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)

Introduced in R2006a

updateFileSeparator

Update file separator character for file lists in model build information

Syntax

```
updateFileSeparator(buildinfo,separator)
```

Description

`updateFileSeparator(buildinfo,separator)` changes instances of the current file separator (/ or \) in the model build information to the specified file separator.

The default value for the file separator matches the value returned by the MATLAB command `filesep`. For template makefile (TMF) approach builds, you can override the default by defining a separator with the `MAKEFILE_FILESEP` macro in the template makefile (see “Cross-Compile Code Generated on Microsoft Windows” (Simulink Coder). If the `GenerateMakefile` parameter is set, the code generator overrides the default separator and updates the model build information after evaluating the `PostCodeGenCommand` configuration parameter.

Examples

Update File Separator in Build Information

Update object `myModelBuildInfo` to apply the Windows file separator.

```
myModelBuildInfo = RTW.BuildInfo;
updateFileSeparator(myModelBuildInfo,'\');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`separator` — File separator character for path specifications in the build information
`'\' | '/'`

The separator argument specifies the file separator `\` (Windows) or `/` (UNIX) to use in file path specifications in the build information.

Example: '\'

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFilePathsAndExtensions`

Topics

“Customize Post-Code-Generation Build Processing” (Simulink Coder)
“Cross-Compile Code Generated on Microsoft Windows” (Simulink Coder)

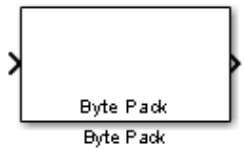
Introduced in R2006a

3

Blocks in Embedded Coder—Alphabetical List

Byte Pack

Convert input signals to `uint8` vector



Library

Embedded Coder/Embedded Targets/Host Communication

Description

Using the input port, the block converts data of one or more data types into a single `uint8` vector for output. With the options available, you specify the input data types and the alignment of the data in the output vector. Because UDP messages are in `uint8` data format, use this block before a UDP Send block to format the data for transmission using the UDP protocol.

Parameters

Input port data types (cell array)

Specify the data types for the different signals as part of the parameters. The block supports all Simulink data types except characters. Enter the data types as Simulink types in the cell array, such as `'double'` or `'int32'`. The order of the data type entries in the cell array must match the order in which the data arrives at the block input. This block determines the signal sizes automatically. The block has at least one input port and only one output port.

Byte alignment

This option specifies how to align the data types to form the `uint8` output vector. Select one of the values in bytes from the list.

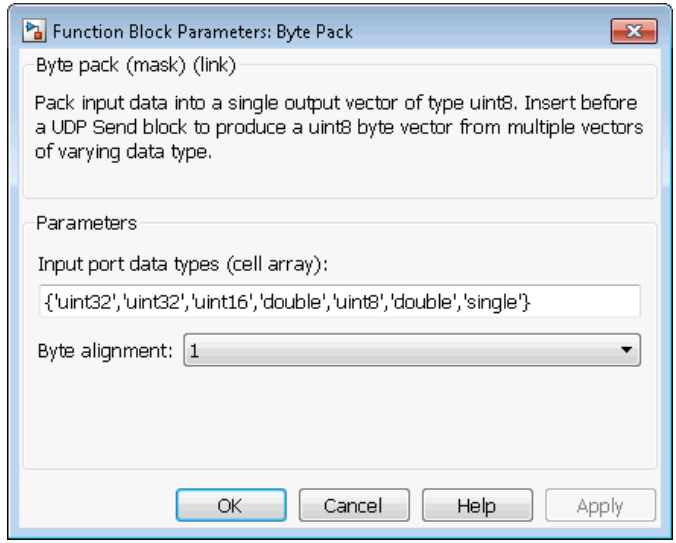
Alignment can occur on 1, 2, 4, or 8-byte boundaries depending on the value you choose. The value defaults to 1. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithms that each element in the output vector begins on a byte boundary specified by the alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

Selecting 1 for **Byte alignment** provides the tightest packing, without holes between data types in the various combinations of data types and signals.

Sometimes, you can have multiple data types of varying lengths. In such cases, specifying a 2-byte alignment can produce 1–byte gaps between uint8 or int8 values and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time. You can specify data alignment options with data types.

Example

Use a cell array to enter input data types in the **Input port data types** parameter. The order of the data types you enter must match the order of the data types at the block input.



In the cell array, you provide the order in which the block expects to receive data—`uint32`, `uint32`, `uint16`, `double`, `uint8`, `double`, and `single`. With this information, the block automatically provides the number of block inputs.

Byte alignment equal to 2 specifies that each new value begins 2 bytes from the previous data boundary.

The example shows the following data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

When the signals are scalar values (not matrices or vectors in this example), the first signal value in the vector starts at 0 bytes. Then, the second signal value starts at 2 bytes, and the third at 4 bytes. Next, the fourth signal value follows at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. As the example shows, the packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

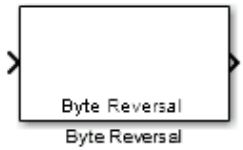
See Also

Byte Reversal, Byte Unpack

Introduced in R2011a

Byte Reversal

Reverse order of bytes in input word



Library

Embedded Coder/Embedded Targets/Host Communication

Description

Byte reversal changes the order of the bytes in data you input to the block. Use this block when your process communicates between targets that use different endianness, such as between Intel® processors that are little endian and others that are big endian. Texas Instruments™ processors are little-endian by default.

To exchange data with a processor that has different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

Parameters

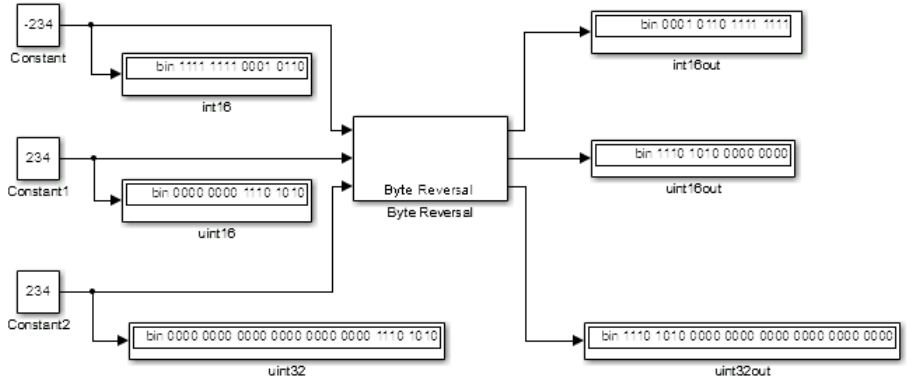
Number of inputs

Specify the number of block inputs. The number of block inputs adjusts automatically to match value so the number of outputs equals the number of inputs.

When you use more than one input port, each input port maps to the matching output port. Data entering input port 1 leaves through output port 1, and so on.

Reversing the bytes does not change the data type. Input and output retain matching data type.

The following model shows byte reversal in use. In this figure, the input and output ports match for each path.



See Also

Byte Pack, Byte Unpack

Introduced in R2011a

Byte Unpack

Unpack UDP `uint8` input vector into Simulink data type values

Library

Embedded Coder/Embedded Targets//Host Communication

Description

Byte Unpack is the inverse of the Byte Pack block. It takes a UDP message from a UDP receive block as a `uint8` vector, and outputs Simulink data types in various sizes depending on the input vector.

The block supports all Simulink data types.

Parameters

Output port dimensions (cell array)

Containing a cell array, each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding Byte Pack block in the model. Entering one value means that the block applies that dimension to all data types.

Output port data types (cell array)

Specify the data types for the different input signals to the Pack block. The block supports all Simulink data types—`single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`, and `Boolean`. The entry here is the same as the Input port data types parameter in the Byte Pack block in the model. You can enter one data type and the block applies that type to all output ports.

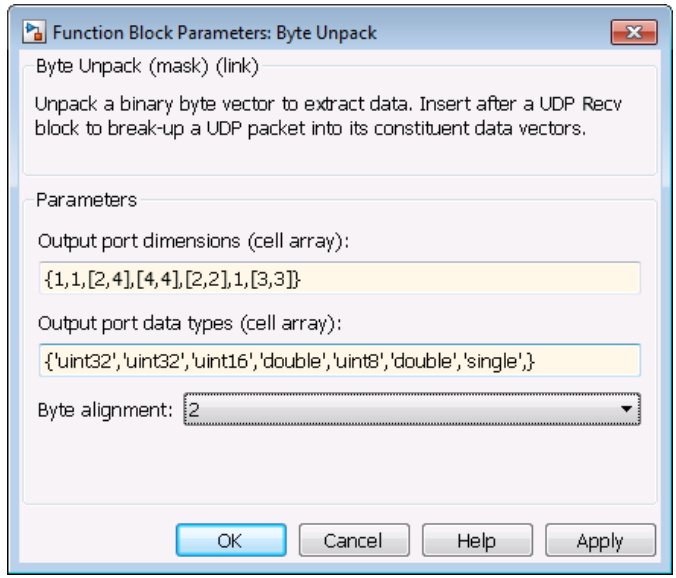
Byte Alignment

This option specifies how to align the data types to form the input `uint8` vector. Match this setting with the corresponding Byte Pack block alignment value of 1, 2, 4, or 8 bytes.

Example

This figure shows the Byte Unpack block that corresponds to the example in the Byte Pack example. The **Output port data types (cell array)** entry shown is the same as the **Input port data types (cell array)** entry in the Byte Pack block

`{'uint32','uint32','uint16','double','uint8','double','single'}`.



In addition, the **Byte alignment** setting matches as well. **Output port dimensions (cell array)** now includes scalar values and matrices to show how to enter nonscalar values. The example for the Byte Pack block assumed only scalar inputs.

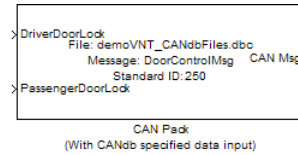
See Also

Byte Pack, Byte Reversal

Introduced in R2011a

CAN Pack

Pack individual signals into CAN message



Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

Description

The CAN Pack block loads signal data into a message at specified intervals during the simulation.

Note To use this block, you also need a license for Simulink software.

CAN Pack block has one input port by default. The number of block inputs is dynamic and depends on the number of signals you specify for the block. For example, if your block has four signals, it has four block inputs.

This block has one output port, CAN Msg. The CAN Pack block takes the specified input parameters and packs the signals into a message.

Other Supported Features

The CAN Pack block supports:

- The use of Simulink Accelerator™ Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation to deploy models to targets.

Note Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

For more information on these features, see the Simulink documentation.

Dialog Box

Use the Function Block Parameters dialog box to select your CAN Pack block parameters.

Parameters

Data is input as

Select your data signal:

- **raw data**: Input data as a uint8 vector array. If you select this option, you only specify the message fields. all other signal parameter fields are unavailable. This option opens only one input port on your block.
- **manually specified signals**: Allows you to specify data signal definitions. If you select this option, use the **Signals** table to create your signals. The number of block inputs depends on the number of signals you specify.
- **CANdb specified signals**: Allows you to specify a CAN database file that contains message and signal definitions. If you select this option, select a CANdb file. The number of block inputs depends on the number of signals specified in the CANdb file for the selected message.

Note The block supports the following input signals data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** list. Click **Browse** to find the CANdb file on your system. The

message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message.

Note File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

Message list

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** field and you select a CANdb file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

Message

Name

Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to input raw data or manually specify signals. This option in unavailable if you choose to use signals from a CANdb file.

Identifier type

Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to input raw data or manually specify signals. For `CANdb specified signals`, the **Identifier type** inherits the type from the database.

Identifier

Specify your CAN message ID. This number must be a positive integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to input raw data or manually specify signals.

Length (bytes)

Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your data input, the CANdb file defines the length of your message. If not, this field defaults to `8`. This option is available if you choose to input raw data or manually specify signals.

Remote frame

Specify the CAN message as a remote frame.

Output as bus

Select this option for the block to output CAN messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Signals” (Simulink).

Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. The start bit must be an integer from 0 through 63.

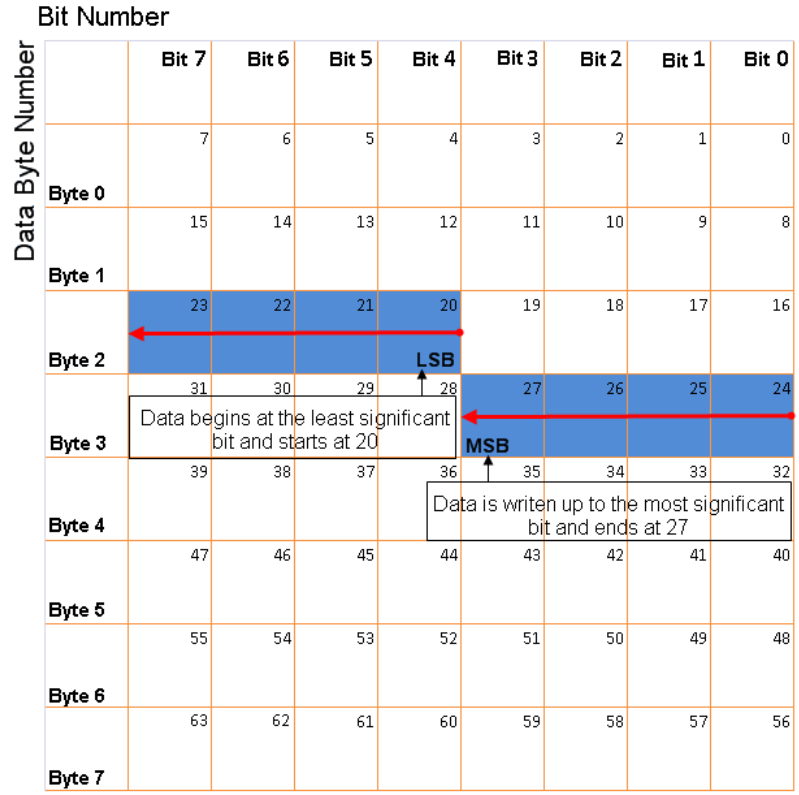
Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order

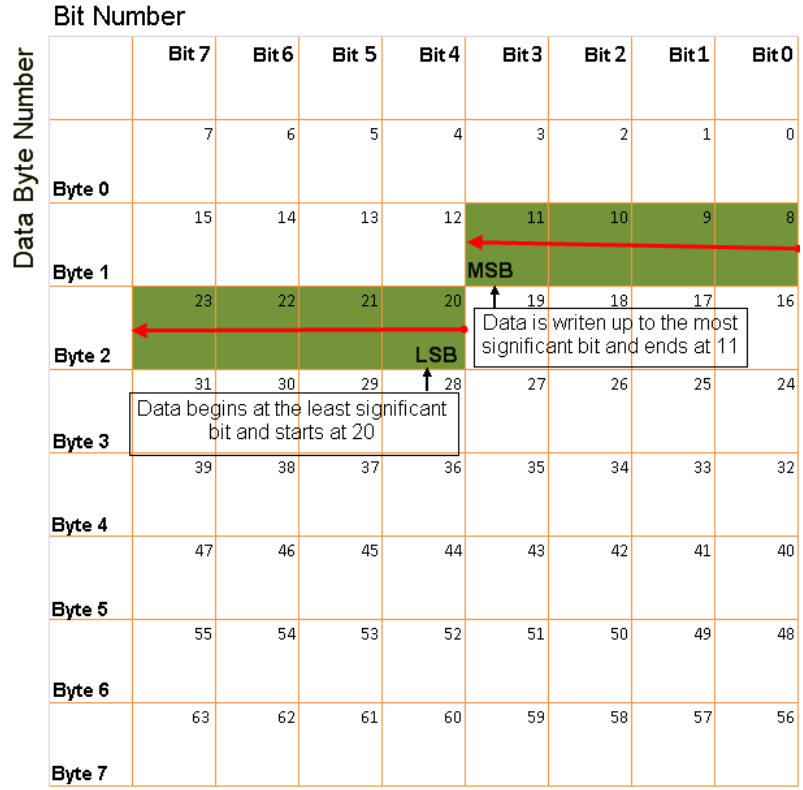
Select either of the following options:

- LE: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.



Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where byte order is in big-endian format (Motorola®). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Multiplex type

Specify how the block packs the signals into the CAN message at each timestep:

- `Standard`: The signal is packed at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is packed. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is packed if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	N/A
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	N/A

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block packs Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block packs Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that timestep.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to pack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 3-17 to understand how physical values are converted to raw values packed into a message.

Offset

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 3-17 to understand how physical values are converted to raw values packed into a message.

Min, **Max**

Define a range of signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

Conversion Formula

The conversion formula is

```
raw_value = (physical_value - Offset) / Factor
```

where `physical_value` is the original value of the signal, and `raw_value` is the packed signal value.

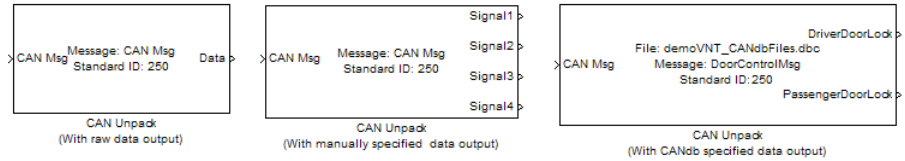
See Also

Blocks
CAN Unpack

Introduced in R2009a

CAN Unpack

Unpack individual signals from CAN messages



Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

Description

The CAN Unpack block unpacks a CAN message into signal data using the specified output parameters at every timestep. Data is output as individual signals.

Note To use this block, you also need a license for Simulink software.

The CAN Unpack block has one output port by default. The number of output ports is dynamic and depends on the number of signals you specify for the block to output. For example, if your block has four signals, it has four output ports.

Other Supported Features

The CAN Unpack block supports:

- The use of Simulink Accelerator Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation to deploy models to targets.

Note Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32 bits long.

For more information on these features, see the Simulink documentation.

Dialog Box

Use the Function Block Parameters dialog box to select your CAN message unpacking parameters.

Parameters

Data to be output as

Select your data signal:

- **raw data**: Output data as a uint8 vector array. If you select this option, you only specify the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.
- **manually specified signals**: Allows you to specify data signals. If you select this option, use the `Signals` table to create your signals message manually.

  The number of output ports on your block depends on the number of signals you specify. For example, if you specify four signals, your block has four output ports.
- **CANdb specified signals**: Allows you to specify a CAN database file that contains data signals. If you select this option, select a CANdb file.

  The number of output ports on your block depends on the number of signals specified in the CANdb file. For example, if the selected message in the CANdb file has four signals, your block has four output ports.

Note For manually or CANdb specified signals, the default output signal data type is double. To specify other types, use a Signal Specification block. This allows the block to

support the following output signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point types.

CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data to be output as** list. Click **Browse** to find the CANdb file on your system. The messages and signal definitions specified in the CANdb file populate the **Message** section of the dialog box. The signals specified in the CANdb file populate **Signals** table.

Note File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

Message list

This option is available if you specify that your data is to be output as a CANdb file in the **Data to be output as** list and you select a CANdb file in the **CANdb file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

Message

Name

Specify a name for your CAN message. The default is `CAN Msg`. This option is available if you choose to output raw data or manually specify signals.

Identifier type

Specify whether your CAN message identifier is a `Standard` or an `Extended` type. The default is `Standard`. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to output raw data or manually specify signals. For CANdb-specified signals, the **Identifier type** inherits the type from the database.

Identifier

Specify your CAN message ID. This number must be a integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If

you specify `-1`, the block unpacks the messages that match the length specified for the message. You can also specify hexadecimal values using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

Length (bytes)

Specify the length of your CAN message from 0 to 8 bytes. If you are using `CANdb specified signals` for your output data, the CANdb file defines the length of your message. If not, this field defaults to `8`. This option is available if you choose to output raw data or manually specify signals.

Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message. The start bit must be an integer from 0 through 63.

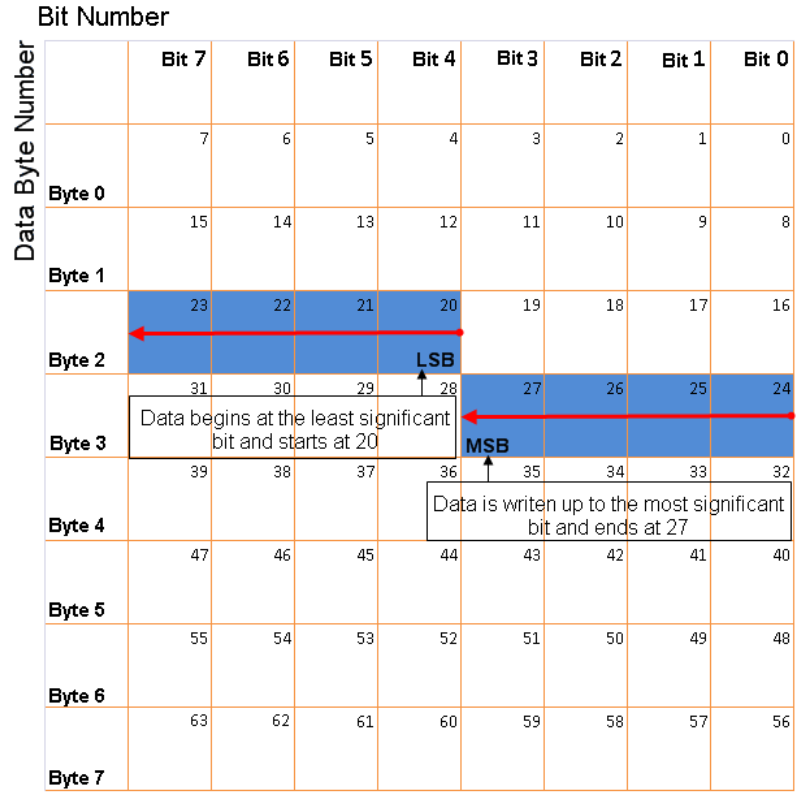
Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order

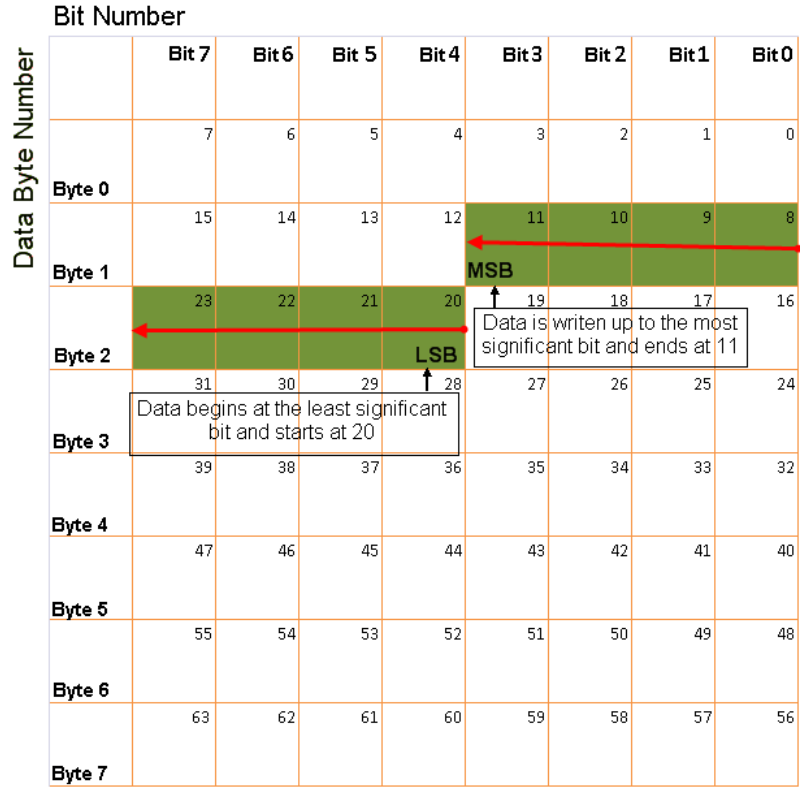
Select either of the following options:

- `LE`: Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.



Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.



Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- `signed` (default)
- `unsigned`
- `single`
- `double`

Multiplex type

Specify how the block unpacks the signals from the CAN message at each timestep:

- `Standard`: The signal is unpacked at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is unpacked. You can specify only one `Multiplexor` signal per message.
- `Multiplexed`: The signal is unpacked if the value of the `Multiplexor` signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	N/A
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	N/A

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block unpacks Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block unpacks Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that timestep.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be `Multiplexed`. The value you provide here must match the `Multiplexor` signal value at run time for the block to unpack the `Multiplexed` signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). See “Conversion Formula” on page 3-26 to understand how unpacked raw values are converted to physical values.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. See “Conversion Formula” on page 3-26 to understand how unpacked raw values are converted to physical values.

Min, **Max**

Define a range of raw signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For **CANdb specified signals**, these settings are read from the CAN database. For **manually specified signals**, you can specify the minimum and maximum physical value of the signal. By default, these settings do not clip signal values that exceed them.

Output Ports

Selecting an **Output ports** option adds an output port to your block.

Output identifier

Select this option to output a CAN message identifier. The data type of this port is **uint32**.

Output remote

Select this option to output the message remote frame status. This option adds a new output port to the block. The data type of this port is **uint8**.

Output timestamp

Select this option to output the message time stamp. This option adds a new output port to the block. The data type of this port is **double**.

Output length

Select this option to output the length of the message in bytes. This option adds a new output port to the block. The data type of this port is **uint8**.

Output error

Select this option to output the message error status. This option adds a new output port to the block. An output value of `1` on this port indicates that the incoming message is an error frame; otherwise the output value is `0`. The data type of this port is **uint8**.

Output status

Select this option to output the message received status. The status is `1` if the block receives new message and `0` if it does not. This option adds a new output port to the block. The data type of this port is **uint8**.

If you do not select an **Output ports** option, the number of output ports on your block depends on the number of signals you specify.

Conversion Formula

The conversion formula is

```
physical_value = raw_value * Factor + Offset
```

where `raw_value` is the unpacked signal value and `physical_value` is the scaled signal value.

See Also

Blocks
CAN Pack

Introduced in R2009a

Custom MATLAB file

Update active configuration parameters of parent model by using file containing custom MATLAB code

Library: Embedded Coder / Configuration Wizards

Description

When you add a Custom MATLAB file block to your Simulink model and double-click it, a custom MATLAB script executes and configures model parameters that are relevant to code generation. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Use the example MATLAB script, *`matlabroot`*`/toolbox/rtw/rtw/rtwsampleconfig.m` with the Custom MATLAB file block and adapt to your model requirements. The block and the script provide a starting point for customization. For more information, see “Create a Custom Configuration Wizard Block”.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

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

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script
`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with this Configuration Wizards block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

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

Topics

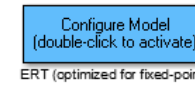
“Configure and Optimize Model with Configuration Wizard Blocks”

Introduced in R2011a

ERT (optimized for fixed-point)

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

Library: Embedded Coder / Configuration Wizards



Description

When you add an ERT (optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point code generation with the ERT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

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

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

Topics

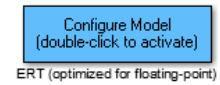
“Configure and Optimize Model with Configuration Wizard Blocks”

Introduced in R2006b

ERT (optimized for floating-point)

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

Library: Embedded Coder / Configuration Wizards



Description

When you add an ERT (optimized for floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for floating-point code generation with the ERT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

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

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for fixed-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point)

Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

Introduced in R2006b

GRT (debug for fixed/floating-point)

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

Library: Embedded Coder / Configuration Wizards

GRT (debug for fixed/floating-point)

Description

When you add a GRT (debug for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point or floating-point code generation, with TLC debugging options enabled, with the GRT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

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

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (optimized for fixed/floating-point)

Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

Introduced in R2006b

GRT (optimized for fixed/floating-point)

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

Library: Embedded Coder / Configuration Wizards

Description

When you add a GRT(optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and optimally configures the model parameters for fixed-point or floating-point code generation with the GRT target. Set a block parameter to invoke the build process after configuring the model.

After you double-click the block, open the Configuration Parameters dialog box to see that the model parameter values have changed.

Note To provide a quick way to switch between configurations, you can include more than one Configuration Wizard block in your model.

Parameters

`Configure the model for` — Configuration objective

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

Objective for updating the active configuration parameters of a parent model by using a predefined MATLAB script.

`Configuration function` — Custom MATLAB script

`rtwsampleconfig` (default)

Predefined or custom MATLAB script that updates the active configuration parameters of the parent model.

Dependencies

This parameter is only used with the Custom MATLAB file block. To enable this parameter, set **Configure the model for** to `Custom`.

`Invoke build process after configuration` — Initiate build process
`off` (default) | `on`

If selected, the predefined script initiates the code generation and build process after updating the configuration parameters.

See Also

Custom MATLAB file | ERT (optimized for fixed-point) | ERT (optimized for floating-point) | GRT (debug for fixed/floating-point)

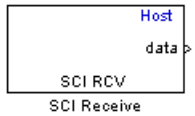
Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

Introduced in R2006b

Host SCI Receive

Configure host-side serial communications interface to receive data from serial port



Library

Embedded Coder/ Embedded Targets/ Host Communication

Description

Specify the configuration of data being received from the target by this block.

The data package being received is limited to 16 bytes of ASCII characters, including package headers and terminators. Calculate the size of a package by including the package header, or terminator, or both, and the data size.

Acceptable data types are single, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The number of bytes in each data type is listed in the following table:

Data Type	Byte Count
`single`	4 bytes
`int8` and `uint8`	1 byte
`int16` and `uint16`	2 bytes
`int32` and`uint32`	4 bytes

For example, if your data package has package header 'S' (1 byte) and package terminator 'E' (1 byte), that leaves 14 bytes for the actual data. If your data is of type `int8`, there is room in the data package for 14 `int8`s. If your data is of type `uint16`, there is room in the data package for 7 `uint16`s. If your data is of type `int32`, there is room in the data package for only 3 `int32`s, with 2 bytes left over. Even though you could fit two `int8`s or one `uint16` in the remaining space, you may not, because you cannot mix data types in the same package.

The number of data types that can fit into a data package determine the data length (see **Data length** in the Dialog Box description). In the example just given, the 14 for data type `int8` and the 7 for data type `uint16` are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run time errors, may result.

Parameters

Port name

You may configure up to four COM ports (COM1 through COM4) for up to four host-side SCI Receive blocks.

Additional package header

This field specifies the data located at the front of the received data package, which is not part of the data being received, and generally indicates start of data. The additional package header must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not received nor are they included in the total byte count.

Note Match additional package headers or terminators with those specified in the target SCI transmit block.

Additional package terminator

This field specifies the data located at the end of the received data package, which is not part of the data being received, and generally indicates end of data. The additional package terminator must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not received nor are they included in the total byte count.

Data type

Choice of single, int8, uint8, int16, uint16, int32, or uint32.

The input port of the SCI Transmit block accepts only one of these values. Which value it accepts is inherited from the data type from the input (the data length is also inherited from the input). Data must consist of only one data type; you cannot mix types.

Data length

How many of **Data type** the block receives (not bytes). Anything more than 1 is a vector. The data length is inherited from the input (the data length input to the SCI Transmit block).

Initial output

Default value from the SCI Receive block. This value is used, for example, if a connection time-out occurs and the **Action taken when connection timeout** field is set to “Output the last received value”, but nothing yet has been received.

Action Taken when connection times out

Specify what to output if a connection time-out occurs. If “Output the last received value” is selected, the block outputs the last received value. If a value has not been received, the block outputs the **Initial output**.

If you select `Output custom value`, use the **Output value when connection times out** field to set the custom value.

Sample time

Determines how often the SCI Receive block is called (in seconds). When you set this value to `-1`, the model inherits the sample time value of the model. To execute this block asynchronously, set **Sample Time** to `-1`.

Output receiving status

Selecting this check box creates a **Status** block output that provides the status of the transaction.

The error status may be one of the following values:

- `0`: No errors
- `1`: A time-out occurred while the block was waiting to receive data
- `2`: There is an error in the received data (checksum error)
- `3`: SCI parity error flag — Occurs when a character is received with a mismatch
- `4`: SCI framing error flag — Occurs when an expected stop bit is not found

See Also

Host SCI Transmit

Introduced in R2011a

Host SCI Setup

Configure COM ports for host-side SCI Transmit and Receive blocks

Library

Embedded Coder/ Embedded Targets/ Host Communication

Description

Standardize COM port settings for use by the host-side SCI Transmit and Receive blocks. Setting COM port configurations globally with the SCI Setup block avoids conflicts (e.g., the host-side SCI Transmit block cannot use COM1 with settings different than those the COM1 used by the host-side SCI Receive block) and requires that you set configurations only once for each COM port. The SCI Setup block is a stand alone block.

Parameters

Communication Mode

Raw data or protocol. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlocks do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

If you specify protocol mode, some handshaking between host and target occurs. The transmitting side sends $SND indicating that it is ready to transmit. The receiving side sends back $RDY indicating that it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include

- Data is received as expected (checksum)
- Data is received by target
- Time consistency; each side waits for its turn to send or receive

Note Deadlocks can occur if one SCI Transmit block is trying to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

Baud rate

Choose from 110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, or 115200.

Number of stop bits

Select 1 or 2.

Parity mode

Select none, odd, or even.

Timeout

Enter values greater than or equal to 0, in seconds. When the COM port involved is using protocol mode, this value indicates how long the transmitting side waits for an acknowledgement from the receiving side or how long the receiving side waits for data. The system displays a warning message if the time-out is exceeded, every *n* number of seconds, *n* being the value in **Timeout**.

Note Simulink suspends processing for the length of the time-out. During that time you cannot perform actions in Simulink. If the time-out is set for a long period of time, it may appear that Simulink has frozen.

See Also

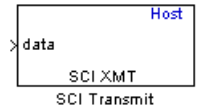
Host SCI Receive

Host SCI Transmit

Introduced in R2011a

Host SCI Transmit

Configure host-side serial communications interface to transmit data to serial port



Library

Embedded Coder/ Embedded Targets/ Host Communication

Description

Specify the configuration of data being transmitted to the target from this block.

The data package being sent is limited to 16 bytes of ASCII characters, including package headers and terminators. Calculate the size of a package by figuring in package header, or terminator, or both, and the data size.

Acceptable data types are `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The byte size of each data type is as follows:

Data Type	Byte Count
`single`	4 bytes
`int8` & `uint8`	1 byte
`int16` & `uint16`	2 bytes
`int32` & `uint32`	4 bytes

For example, if your data package has package header “S” (1 byte) and package terminator “E” (1 byte), that leaves 14 bytes for the actual data. If your data is of type `int8`, there is room in the data package for 14 `int8`s. If your data is of type `uint16`, there is room in the data package for only 7 `uint16`s. If your data is of type `int32`, there is room in the data package for only 3 `int32`s, with 2 bytes left over. Even though you could fit two `int8`s or one `uint16` in the remaining space, you may not, because you cannot mix data types in the same package.

The number of data types that can fit into a data package determine the data length (see **Data length** in the Dialog Box description). In the example just given, the 14 for data type int8 and the 7 for data type uint16 are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run time errors, may result.

Parameters

Port name

You may configure up to four COM ports (COM1 through COM4) for up to four host-side SCI Transmit blocks.

Additional package header

This field specifies the data located at the front of the transmitted data package, which is not part of the data being transmitted, and generally indicates start of data. The additional package header must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not sent nor are they included in the total byte count.

Note Match additional package headers or terminators with those specified in the target SCI receive block.

Additional package terminator

This field specifies the data located at the end of the transmitted data package, which is not part of the data being sent, and generally indicates end of data. The additional package terminator must be an ASCII value. You can use text or a number (0–255). You must put single quotes around text entered in this field, but the quotes are not transmitted nor are they included in the total byte count.

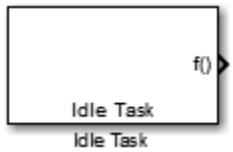
See Also

Host SCI Receive

Introduced in R2011a

Idle Task

Create free-running task



Description

The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. The tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

This block is not supported on targets running an operating system or RTOS.

Vectorized Output

The block output comprises a set of vectors—the task numbers vector and the preemption flag or flags vector. A preemption-flag vector must be the same length as the number of tasks vector unless the preemption flag vector has only one element. The value of the preemption flag determines whether a given interrupt (and task) is preemptible. Preemption overrides prioritization. A lower-priority nonpreemptible task can preempt a higher-priority preemptible task.

When the preemption flag vector has one element, that element value applies to the functions in the downstream subsystem as defined by the task numbers in the task number vector. If the preemption flag vector has the same number of elements as the task number vector, each task defined in the task number vector has a preemption status defined by the value of the corresponding element in the preemption flag vector.

Parameters

Task numbers

Identifies the created tasks by number. Enter as many tasks as you need by entering a vector of integers. The default values are `[1,2]` to indicate that the downstream subsystem has two functions.

The values you enter determine the execution order of the functions in the downstream subsystem, while the number of values you enter corresponds to the number of functions in the downstream subsystem.

Enter a vector containing the same number of elements as the number of functions in the downstream subsystem. This vector can contain up to 16 elements, and the values must be from 0 to 15 inclusive.

The value of the first element in the vector determines the order in which the first function in the subsystem is executed, the value of the second element determines the order in which the second function in the subsystem is executed, and so on.

For example, entering [2,3,1] in this field indicates that there are three functions to be executed, and that the third function is executed first, the first function is executed second, and the second function is executed third. After the functions are executed, the Idle Task block cycles back and repeats the execution of the functions in the same order.

Preemption flags

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering `1` indicates that the interrupt can be preempted. Entering `0` indicates the interrupt cannot be preempted. When **Task numbers** contains more than one task, you can assign different preemption flags to each task by entering a vector of flag values, corresponding to the order of the tasks in **Task numbers**. If **Task numbers** contains more than one task, and you enter only one flag value here, that status applies to the tasks.

In the default settings `[0 1]`, the task with priority `1` in **Task numbers** is not preemptible, and the priority `2` task can be preempted.

Enable simulation input

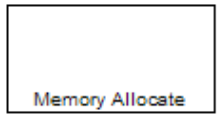
When you select this option, Simulink software adds an input port to the Idle Task block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Note Select this check box to test asynchronous interrupt processing behavior in Simulink software.

Introduced in R2011a

Memory Allocate

Allocate memory section



Memory Allocate

Description

On C2xxx, C5xxx, or C6xxx processors, this block directs the TI compiler to allocate memory for a new variable you specify. Parameters in the block dialog box let you specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.

The block does not verify whether the entries for your variable are valid, such as checking the variable name, data type, or section. You must check that all variable names are valid, that they use valid data types, and that all section names you specify are valid as well.

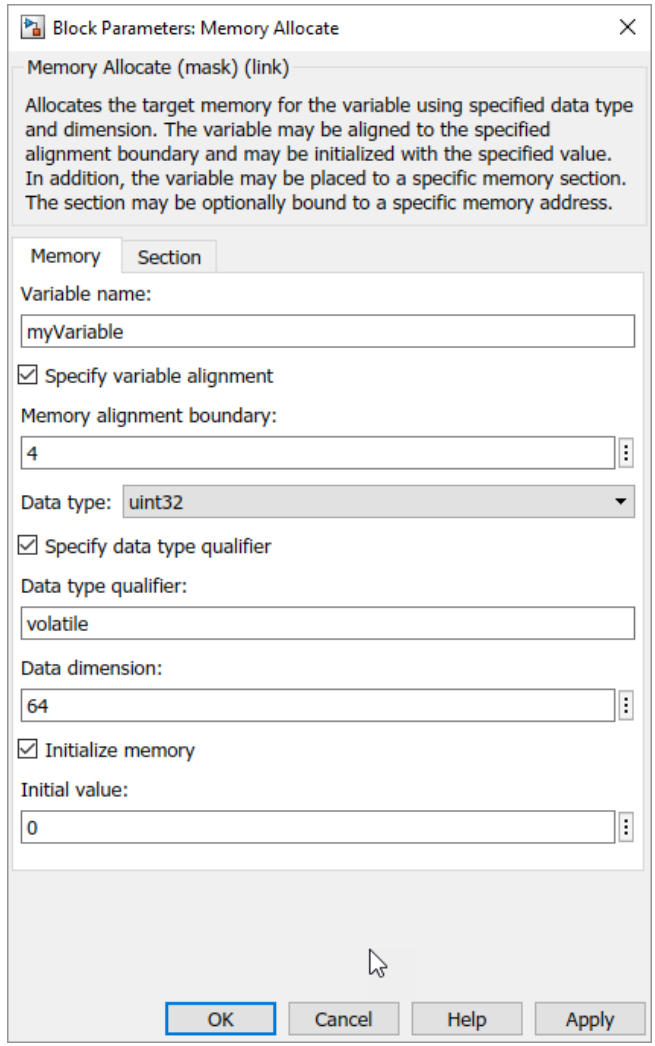
The block does not have input or output ports. It only allocates a memory location. You do not connect it to other blocks in your model.

Dialog Box

The block dialog box comprises multiple tabs:

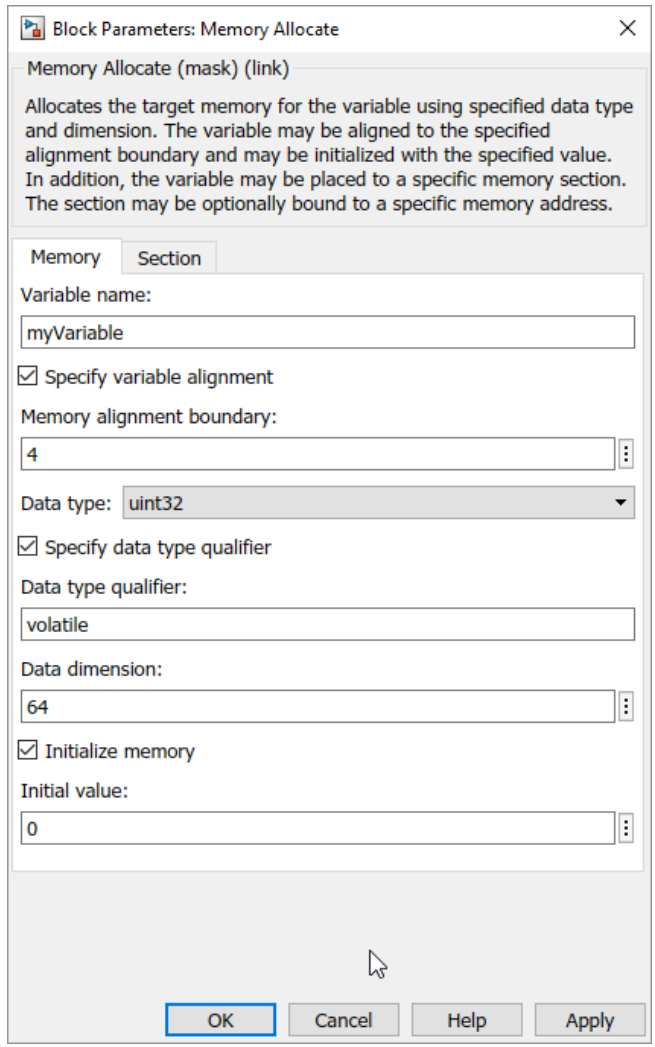
- **Memory** — Allocate the memory for storing variables. Specify the data type and size.
- **Section** — Specify the memory section in which to allocate the variable.

The dialog box images show all of the available parameters enabled. Some of the parameters shown do not appear until you select one or more other parameters.



The following sections describe the contents of each pane in the dialog box.

Memory Parameters



You find the following memory parameters on this tab.

Variable name

Specify the name of the variable to allocate. The variable is allocated in the generated code.

Specify variable alignment

Select this option to direct the compiler to align the variable in **Variable name** to an alignment boundary. When you select this option, the **Memory alignment boundary** parameter appears so you can specify the alignment. Use this parameter and **Memory alignment boundary** when your processor requires this feature.

Memory alignment boundary

After you select **Specify variable alignment**, this option enables you to specify the alignment boundary in bytes. If your variable contains more than one value, such as a vector or an array, the elements are aligned according to rules applied by the compiler.

Data type

Defines the data type for the variable. Select from the list of types available.

Specify data type qualifier

Selecting this enables **Data type qualifier** so you can specify the qualifier to apply to your variable.

Data type qualifier

After you select **Specify data type qualifier**, you enter the desired qualifier here. `Volatile` is the default qualifier. Enter the qualifier you need as text. Common qualifiers are `static` and `register`. The block does not check for valid qualifiers.

Data dimension

Specifies the number of elements of the type you specify in **Data type**. Enter an integer here for the number of elements.

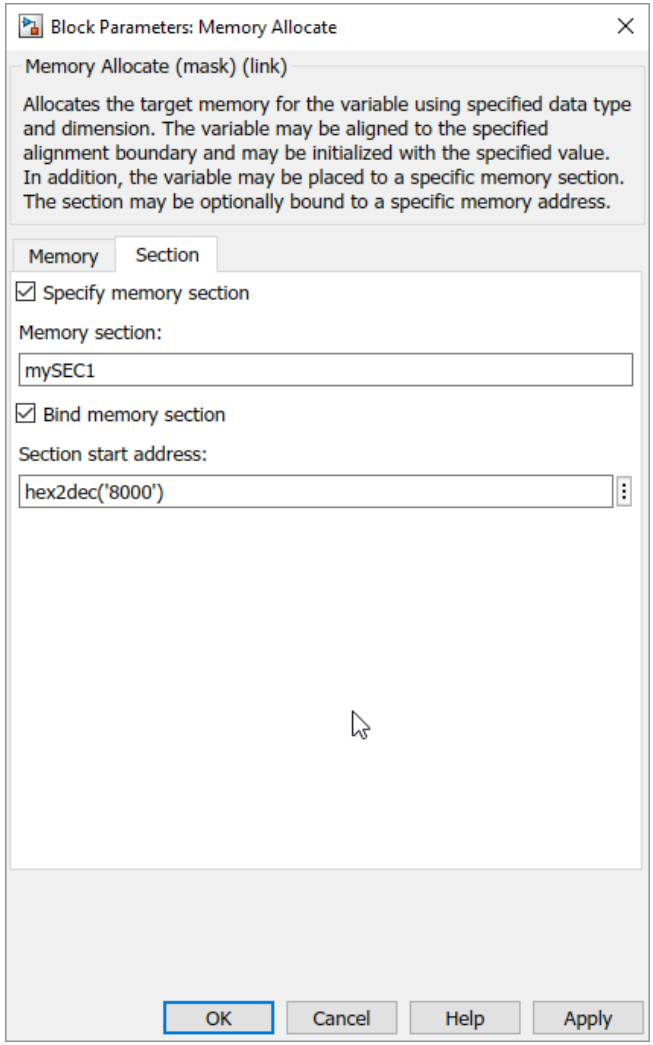
Initialize memory

Directs the block to initialize the memory location to a fixed value before processing.

Initial value

Specifies the initialization value for the variable. At run time, the block sets the memory location to this value.

Section Parameters



Parameters on this pane specify the section in memory to store the variable.

Specify memory section

Selecting this parameter enables you to specify the memory section to allocate space for the variable. Enter either one of the standard memory sections or a custom section that you declare elsewhere in your code.

Memory section

Identify a specific memory section to allocate the variable in **Variable name**. Verify that the section has enough space to store your variable. After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use **Bind memory section** to bind the memory section to a location.

Bind memory section

After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use this parameter to bind the memory section to the location in memory specified in **Section start address**. When you select this, you enable the **Section start address** parameter.

The new memory section specified in **Memory section** is defined when you check this parameter.

Note Do not use **Bind memory section** for existing memory sections.

Section start address

Specify the address to which to bind the memory section. Enter the address in decimal form or in hexadecimal with a conversion to decimal as shown by the default value `hex2dec('8000')`. The block does not verify the address—verify that the address exists and can contain the memory section you entered in **Memory section**.

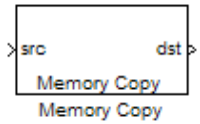
See Also

Memory Copy

Introduced in R2011a

Memory Copy

Copy to and from memory section



Description

In generated code, this block copies variables or data from and to processor memory as configured by the block parameters. Your model can contain as many of these blocks as you require to manipulate memory on your processor.

Each block works with one variable, address, or set of addresses provided to the block. Parameters for the block let you specify both the source and destination for the memory copy, as well as options for initializing the memory locations.

Using parameters provided by the block, you can change options like the memory stride and offset at run time. In addition, by selecting various parameters in the block, you can write to memory at program initialization, at program termination, and at every sample time. The initialization process occurs once, rather than occurring for every read and write operation.

With the custom source code options, the block enables you to add custom ANSI C source code before and after each memory read and write (copy) operation. You can use the custom code capability to lock and unlock registers before and after accessing them. For example, some processors have registers that you may need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

If your processor or board supports quick direct memory access (QDMA) the block provides a parameter to check that implements the QDMA copy operation, and enables you to specify a function call that can indicate that the QDMA copy is finished. Only the C621x, C64xx, and C671x processor families support QDMA copy.

Note Replace Read from Memory and Write To Memory blocks, which were removed in a previous release, with the Memory Copy block.

Block Operations

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. With the options for setting memory initialization and termination, you control when and how the block initializes memory, copies to and from memory, and terminates memory operations. The parameters enable you to turn on and off memory operations in the three periods independently.

Used in combination with the Memory Allocate block, this block supports building custom device drivers, such as PCI bus drivers or codec-style drivers, by letting you manipulate and allocate memory. This block does not require the Memory Allocate block to be in the model.

In a simulation, this block does not perform an operation. The block output is not defined.

Copy Memory

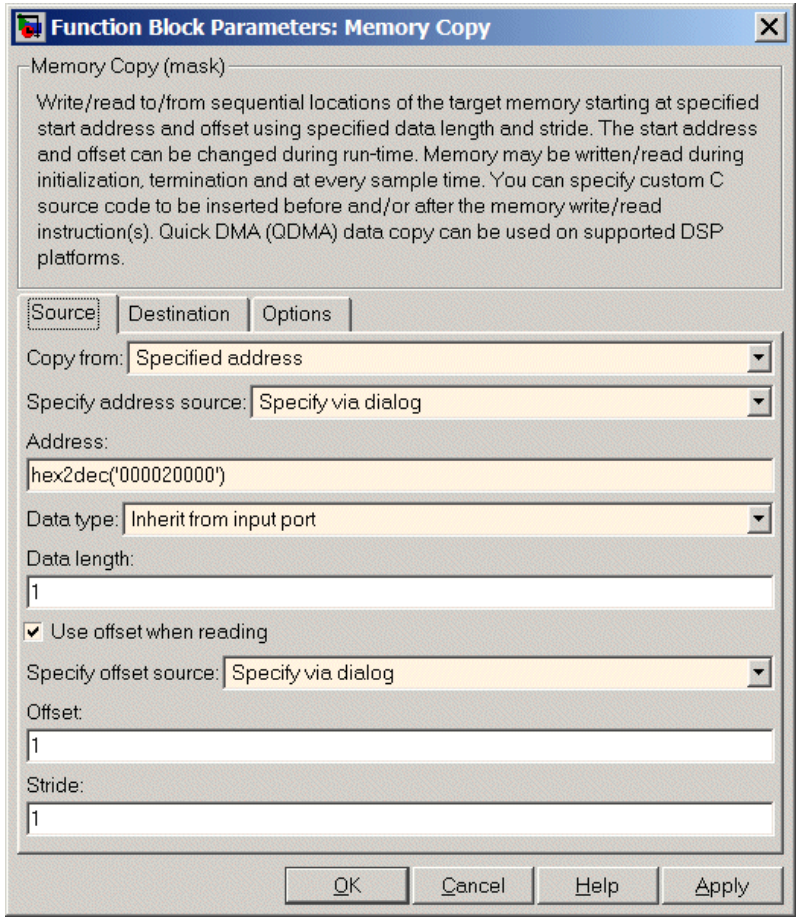
When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

Dialog Box

The block dialog box contains multiple tabs:

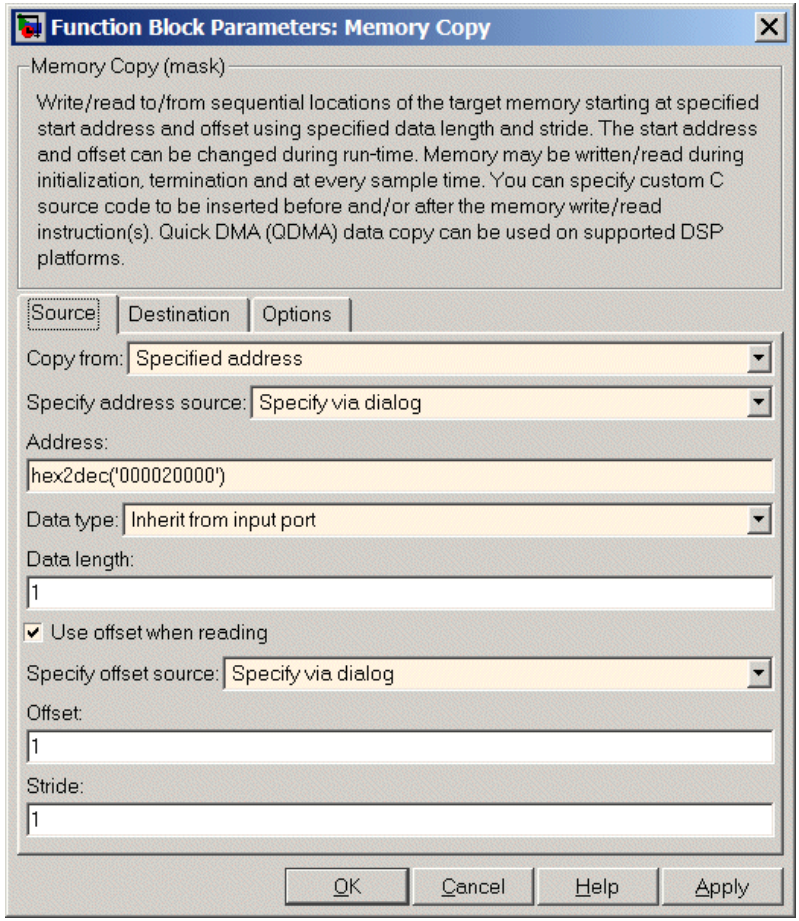
- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various parameters to control the copy process.

The dialog box images show many of the available parameters enabled. Some parameters shown do not appear until you select one or more other parameters. Some parameters are not shown in the figures, but the text describes them and how to make them available.



Sections that follow describe the parameters on each tab in the dialog box.

Source Parameters



Copy from

Select the source of the data to copy. Choose one of the entries on the list:

- `Input port` — This source reads the data from the block input port.
- `Specified address` — This source reads the data at the specified location in **Specify address source** and **Address**.

- `Specified source code symbol` — This source tells the block to read the symbol (variable) you enter in **Source code symbol**. When you select this copy from option, you enable the **Source code symbol** parameter.

  Note If you do not select `Input port` for **Copy from**, change **Data type** from the default `Inherit from source` to one of the data types on the **Data type** list. If you do not make the change, you receive an error message that the data type cannot be inherited because the input port does not exist.

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either `Specify via dialog` or `Input port` from the list. Selecting `Specify via dialog` activates the **Address** parameter for you to enter the address for the variable.

When you select `Input port`, the port label on the block changes to `&src`, indicating that the block expects the address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax. Enter text to specify the symbol exactly as you use it in your code.

Address

When you select Specify via dialog for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal character vector with single quotations marks and use `hex2dec` to convert the address to the expected format. The following example converts `0x1000` to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you could enter either `4096` or `hex2dec('1000')` as the address.

Data type

Use this parameter to specify the type of data that your source uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `Inherit from source` for inheriting the data type from the block input port.

Data length

Specifies the number of elements to copy from the source location. Each element has the data type specified in **Data type**.

Use offset when reading

When you are reading the input, use this parameter to specify an offset for the input read. The offset value is in elements with the assigned data type. The **Specify offset source** parameter becomes available when you check this option.

Specify offset source

The block provides two sources for the offset — `Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when reading the input data.

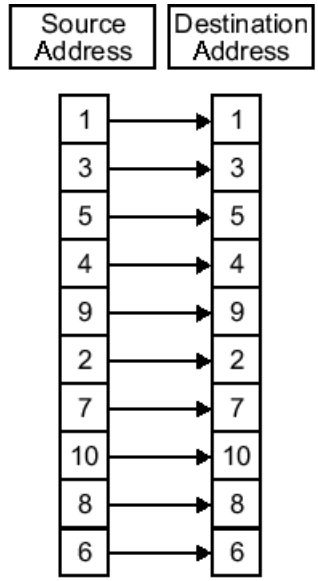
Offset

Offset tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the default value and **Offset** accepts only positive integers of one or greater.

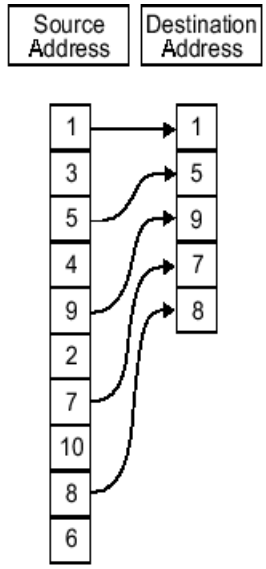
Stride

Stride lets you specify the spacing for reading the input. By default, the stride value is one, meaning the generated code reads the input data sequentially. When you add a stride value that is not equal to one, the block reads the input data elements not sequentially, but by skipping spaces in the source address equal to the stride. **Stride** must be a positive integer.

The next two figures help explain the stride concept. In the first figure you see data copied without a stride. Following that figure, the second figure shows a stride value of two applied to reading the input when the block is copying the input to an output location. You can specify a stride value for the output with parameter **Stride** on the **Destination** pane. Compare stride with offset to see the differences.

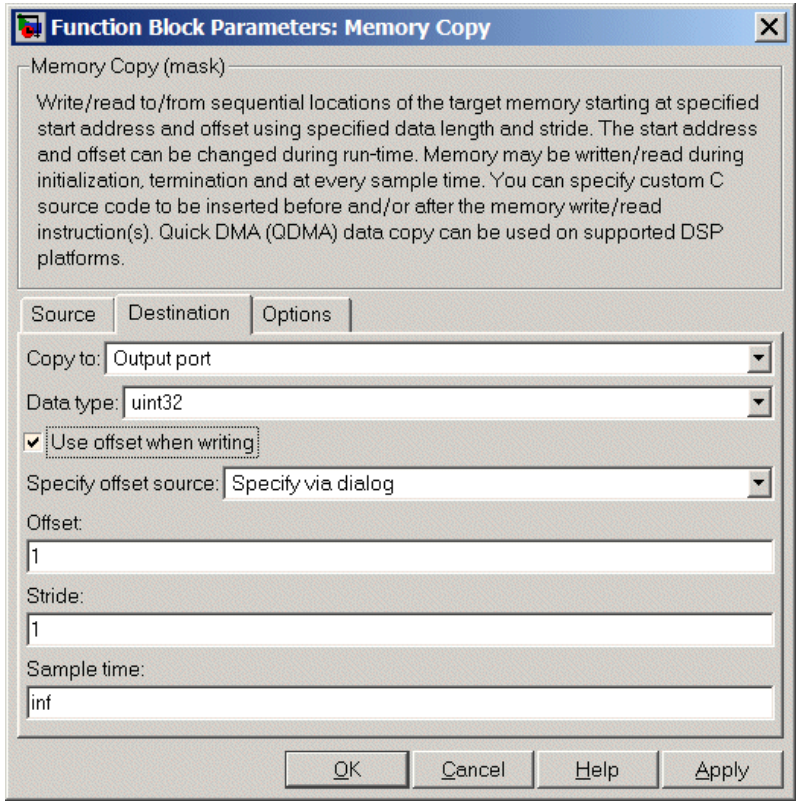


Input Stride = 1
Output Stride = 1
Number of Elements Copied = 10



Input Stride = 2
Output Stride = 1
Number of Elements Copied = 5

Destination Parameters



Copy to

Select the destination for the data. Choose one of the entries on the list:

- `Output port` — Copies the data to the block output port. From the output port the block passes data to downstream blocks in the code.
- `Specified address` — Copies the data to the specified location in **Specify address source** and **Address**.
- `Specified source code symbol` — Tells the block to copy the variable or symbol (variable) to the symbol you enter in **Source code symbol**. When you select this copy to option, you enable the **Source code symbol** parameter.

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either `Specify via dialog` or `Input port` from the list. Selecting `Specify via dialog` activates the **Address** parameter for you to enter the address for the variable.

When you select `Input port`, the port label on the block changes to `&dst`, indicating that the block expects the destination address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax.

Address

When you select Specify via dialog for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal character vector with single quotations marks and use `hex2dec` to convert the address to the expected format. This example converts `0x2000` to decimal form.

```
8192 = hex2dec('2000');
```

For this example, you could enter either `8192` or `hex2dec('2000')` as the address.

Data type

Use this parameter to specify the type of data that your variable uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `inherit from source` for inheriting the data type for the variable from the block input port.

Specify offset source

The block provides two sources for the offset—`Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when writing the output data.

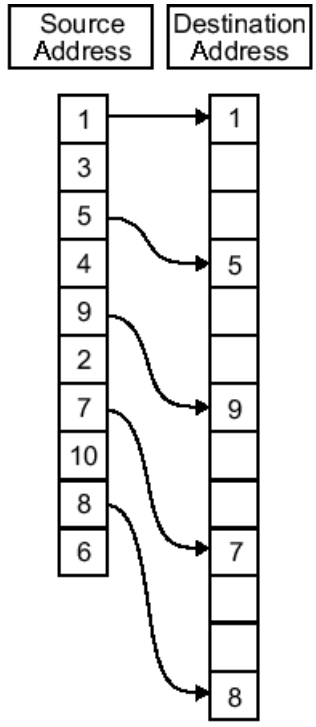
Offset

Offset tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before writing the first value to the destination. One is the default offset value and **Offset** accepts only positive integers of one or greater.

Stride

Stride lets you specify the spacing for copying the input to the destination. By default, the stride value is one, meaning the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value not equal to one, the output data is stored not sequentially, but by skipping addresses equal to the stride. **Stride** must be a positive integer.

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.

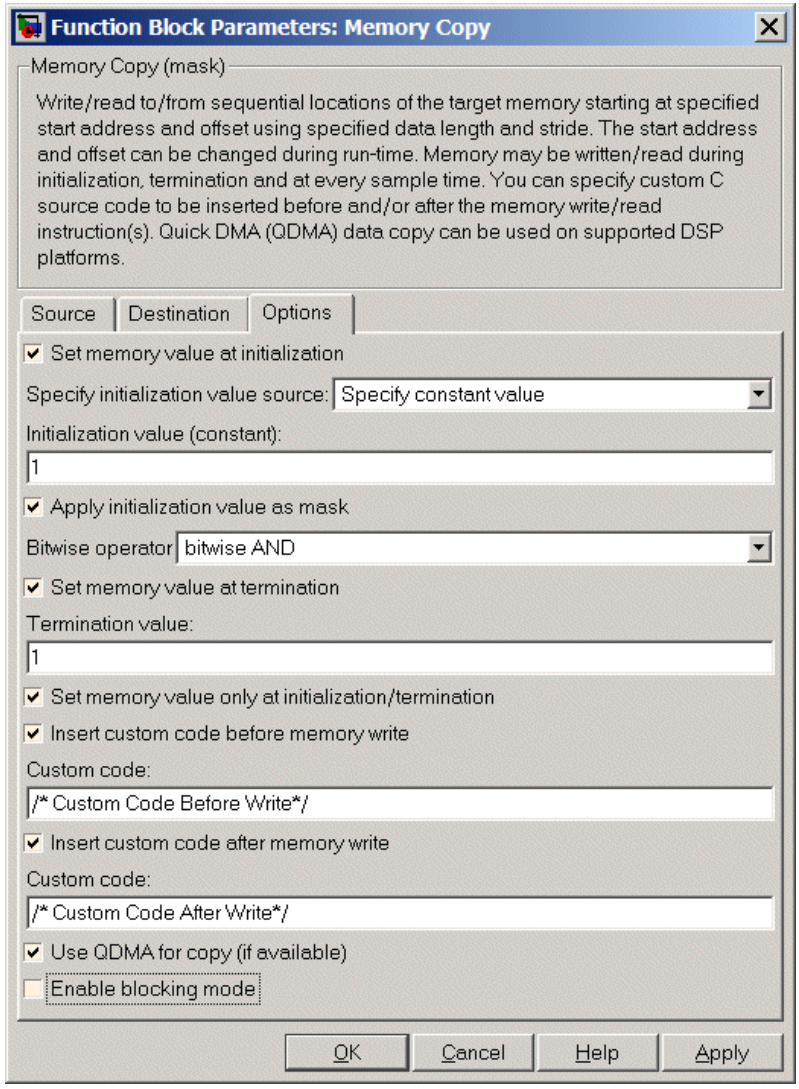


Input Stride = 2
Output Stride = 3
Number of Elements Copied = 5

Sample time

Sample time sets the rate at which the memory copy operation occurs, in seconds. The default value `Inf` tells the block to use a constant sample time. You can set **Sample time** to `-1` to direct the block to inherit the sample time from the input, or from the Simulink software model when there are no block inputs. Enter the sample time in seconds as you need.

Options Parameters



Set memory value at initialization

When you check this option, you direct the block to initialize the memory location to a specific value when you initialize your program at run time. After you select this

option, use the **Set memory value at termination** and **Specify initialization value source** parameters to set your desired value. Alternately, you can tell the block to get the initial value from the block input.

Specify initialization value source

After you check Set memory value at initialization, use this parameter to select the source of the initial value. Choose either

- `Specify constant value` — Sets a single value to use when your program initializes memory.
- `Specify source code symbol` — Specifies a variable (a symbol) to use for the initial value. Enter the symbol as a character vector.

Initialization value (constant)

If you check **Set memory value at initialization** and choose `Specify constant value` for **Specify initialization value source,** enter the constant value to use in this field.

Initialization value (source code symbol)

If you check **Set memory value at initialization** and choose `Specify source code symbol` for **Specify initialization value source,** enter the symbol to use in this field. Use a valid symbol from the symbol table for the program. When you enter the symbol, the block does not verify whether the symbol is a valid one. If it is not valid you get an error when you try to compile, link, and run your generated code.

Apply initialization value as mask

You can use the initialization value as a mask to manipulate register contents at the bit level. Your initialization value is treated as a string of bits for the mask.

Checking this parameter enables the **Bitwise operator** parameter for you to define how to apply the mask value.

To use your initialization value as a mask, the output from the copy has to be a specific address. It cannot be an output port, but it can be a symbol.

Bitwise operator

To use the initialization value as a mask, select one of the entries on the following table from the **Bitwise operator** list to describe how to apply the value as a mask to the memory value.

Bitwise Operator List Entry	Description
`bitwise AND`	Apply the mask value as a bitwise AND to the value in the register.
`bitwise OR`	Apply the mask value as a bitwise OR to the value in the register.
`bitwise exclusive OR`	Apply the mask value as a bitwise exclusive OR to the value in the register.
`left shift`	Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer.
`right shift`	Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer.

Applying a mask to the copy process lets you select individual bits in the result, for example, to read the value of the fifth bit by applying the mask.

Set memory value at termination

Along with initializing memory when the program starts to access this memory location, this parameter directs the program to set memory to a specific value when the program terminates.

Set memory value only at initialization/termination

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. When you check this option, the block only does the memory initialization and termination processes. It does not perform copies during real-time operations.

Insert custom code before memory write

Select this parameter to add custom ANSI C code before the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

Custom code

Enter the custom ANSI C code to insert into the generated code just before the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

Insert custom code after memory write

Select this parameter to add custom ANSI C code immediately after the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

Custom code

Enter the custom ANSI C code to insert into the generated code just after the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

Use QDMA for copy (if available)

For processors that support quick direct memory access (QDMA), select this parameter to enable the QDMA operation and to access the blocking mode parameter.

If you select this parameter, your source and destination data types must be the same or the copy operation returns an error. Also, the input and output stride values must be one.

Enable blocking mode

If you select the **Use QDMA for copy** parameter, select this option to make the memory copy operations blocking processes. With blocking enabled, other processing in the program waits while the memory copy operation finishes.

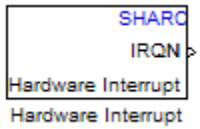
See Also

Memory Allocate

Introduced in R2011a

SHARC Hardware Interrupt

Generate Interrupt Service Routine



Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices® SHARC®/ Scheduling

Description

Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that either run the processes that are downstream from this block or trigger an Idle Task block connected to this block.

Parameters

Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The valid ranges are 8-36 and 38-40.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. The values in this field and the preemption flag entries in **Preemption flags: preemptible-1, non-preemptible-0** define how the code and processor handle interrupts during asynchronous scheduler operations.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Code generation requires rate transition code. Refer to Rate Transitions and Asynchronous Blocks (Simulink Coder). The task priority values facilitate absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. Typically, assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

Preemption flags preemptible – 1, non-preemptible – 0

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering `1` indicates that the interrupt can be preempted.
- Entering `0` indicates the interrupt cannot be preempted.

When **Interrupt numbers** contains more than one interrupt value, you can assign different preemption flags to each interrupt by entering a vector of flag values to correspond to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings `[0 1]`, the interrupt with priority `18` in **Interrupt numbers** is not preemptible and the priority `39` interrupt can be preempted.

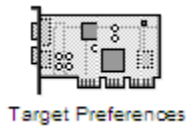
Enable simulation input

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Introduced in R2011a

Target Preferences (Removed)

Configure model for specific IDE, tool chain, board, and processor



Library

Simulink Coder / Desktop Targets

Embedded Coder/ Embedded Targets

Description

The Target Preferences block has been removed from the Simulink block libraries. The contents of the Target Preferences block have been moved to the **Hardware Implementation** pane, located in the Configuration Parameters dialog. For more information, see:

- “Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box”
- “Configure Target Hardware Resources”
- “Hardware Implementation Pane” (Simulink)

Introduced in R2013a

UDP Receive

Receive UDP packet

Block Library

Embedded Coder/Embedded Targets/Host Communication

Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block outputs the contents of a single UDP packet as a data vector. The local IP port number on which the block receives the UDP packets is tunable in the C/C++ generated code.

The generated code for this block relies on prebuilt `.dll` files. You can run this code outside the MATLAB environment, or redeploy it, but you must account for these extra `.dll` files when doing so. The `packNGo` function creates a single zip file containing all of the pieces required to run or rebuild this code. For more details, see “How To Run a Generated Executable Outside MATLAB” (DSP System Toolbox).

Parameters

Local IP port

Specify the IP port number on which to receive UDP packets. This parameter is tunable in the C/C++ generated code but not tunable during simulation. The default is `25000`. The value can be in the range [1 65535].

Note On Linux®, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Remote IP address ('0.0.0.0' to accept all)

Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from other addresses. To accept packets from any IP address, enter `'0.0.0.0'`. This value defaults to `'0.0.0.0'`.

Receive buffer size (bytes)

Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to 8192.

Maximum length for Message

Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of a UDP packet. The system truncates data that exceeds this length. This value defaults to 255.

If you disable **Output variable-size signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

Data type for Message

Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Message is complex

Select this parameter to receive the message as complex data. Clear this parameter if the received message is real. By default, this parameter is not selected.

Output variable-size signal

If your model supports signals of varying length, enable the **Output variable-size signal** parameter. This checkbox defaults to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Blocking time (seconds)

For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note This parameter appears only in the Embedded Coder UDP Receive block.

Sample time (seconds)

Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a smaller value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

Deprecated Parameters

Output port width

Specify the width of packets the block accepts. When you design the transmit end of the UDP communication channel, you decide the packet width. Set this option to a value as large or larger than a packet you expect to receive.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

UDP receive buffer size (bytes)

Specify the size of the buffer to which the system stores UDP packets. The default size is 8192 bytes. Make the buffer large enough to store UDP packets that come in while

your process reads a packet from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Send

Introduced in R2011a

UDP Send

Send UDP message

Block Library

Embedded Coder/Embedded Targets/Host Communication

Description

The UDP Send block transmits an input vector as a UDP message over an IP network port. The remote IP port number to which the block sends the UDP packets is tunable in the C/C++ generated code.

Note Some Simulink blocks and `.exe` files built from models that contain those blocks require shared libraries, such as `.dll` files on Windows. The UDP Send block requires the `networkdevice.dll` library file. To meet this requirement, follow the example on the `packNGo` function page to package the code files for your model. The resulting compressed folder contains the `.dll` files that the model requires, including `networkdevice.dll`. To run this type of `.exe` file outside a MATLAB environment, place the required `.dll` files in the same folder as the `.exe` file, or place them in a folder on the Windows system path. For more details, see “How To Run a Generated Executable Outside MATLAB” (DSP System Toolbox).

Parameters

Remote IP address (`'255.255.255.255'` for broadcast)

Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, `'255.255.255.255'`.

Remote IP port

Specify the port to which the block sends the message. This parameter is tunable in the C/C++ generated code but not tunable during simulation. The default is `25000`. The value can be in the range [1 65535].

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Local IP port source

To let the system automatically assign the port number, select `Automatically determine`. To specify the IP port number using the **Local IP port** parameter, select `Specify via dialog`.

Local IP port

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

Deprecated Parameters

Sample time

Sample time tells the block how long to wait before polling for new messages.

Note This parameter only appears in a deprecated version of the UDP Send block. Replace the deprecated UDP Send block with a current UDP Send block.

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Receive

Introduced in R2011a

DSP/BIOS Hardware Interrupt

Generate Interrupt Service Routine

Library

Embedded Coder Support Package for Texas Instruments C6000™ Processors/ DSP/BIOS

Description

Creates an Interrupt Service Routine (ISR) that executes the task block or subsystem that is downstream from the block. ISRs are functions that the CPU executes in response to an external event.

Interrupt numbers for C6000 family processors range from 0 to 15, with 0 reserved for the reset ISR. The following table presents the set of interrupt numbers for the C6713 processor. For more detailed and specific information about interrupts, refer to Texas Instruments technical documentation for your target processor.

Interrupt Number	Default Event	Module
0	Reset	
1	NMI	
2	Reserved	
3	Reserved	
4	GPINT4	GPIO
5	GPINT5	GPIO
6	GPINT6	GPIO
7	GPINT7	GPIO

Interrupt Number	Default Event	Module
8	EDMAINT	EDMA
9	EMUDTDMA	Emulation
10	SDINT	EMIF
11	EMURTDXRX	Emulation
12	EMURTDXTX	Emulation
13	DSPINT	HPI
14	TINT0	Timer 0
15	TINT1	Timer 1

In models, you usually follow this block with either a DSP/BIOS Task or DSP/BIOS Triggered Task block.

Parameters

Interrupt number(s)

Enter one or more integer values as a vector that represent interrupts. Interrupts have a value from `0`, the highest priority to `15`, lowest priority. As shown, enter the values enclosed in square brackets. For example, entering

```
[3 5 15]
```

results in three interrupt routines. `[5 8]` is the default entry, specifying two interrupts.

Preemption flag(s)

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering `1` indicates that the interrupt can be preempted. Entering `0` indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value here, that status applies to all interrupts.

In the default settings `[0 1]`, the interrupt with priority `5` in **Interrupt numbers** is not preemptible and the priority `8` interrupt can be preempted.

Manage own timer

The ISR generated by this block can manage its own time by reading time from the clock on the board. Selecting this option directs the ISR to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the ISR uses.

Timer resolution (seconds)

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

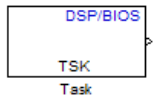
Enable simulation input

Selecting this option adds an input port to the block for simulating inputs in Simulink software. Connect interrupt simulation sources to the input. This option affects simulation only. It does not alter generated code.

Introduced in R2011a

DSP/BIOS Task

Create task that runs as separate DSP/BIOS thread



Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

Description

Creates a free-running task that runs in response to an ISR and as a separate DSP/BIOS™ thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_ert.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

Parameters

Task name (32 characters or less)

Creates a name for the task. Enter up to 32 characters, including numbers and letters. You cannot use the standard C reserved characters, such as / and : in the name.

Task priority (1-15)

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority.

Stack size (bytes)

Specify the size of the stack the task uses. The value defaults to 4096 bytes. Each DSP/BIOS task has a separate stack. This parameter is not related to **System stack size (MAUs)** in the model Configuration Parameters.

Stack memory segment

Specify where the stack resides in memory.

Manage own timer

This block can manage its own time by reading time from the clock on the board. Selecting this option directs the task/block to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the task uses.

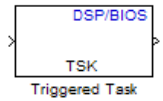
Timer resolution (seconds)

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

Introduced in R2011a

DSP/BIOS Triggered Task

Create asynchronously triggered task



Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

Description

Creates a task that runs asynchronously in response to an ISR and as a separate DSP/BIOS thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

Parameters

Task name (32 characters or less)

Creates a name for the task. Enter up to 32 characters, including numbers and letters. You cannot use the standard C reserved characters, such as / or : in the name.

Task priority (1-15)

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority, unless the preemptible flag (**Preemption flag** option on the DSP/BIOS Hardware Interruptblock) prevents preempting the task.

Stack size (bytes)

Specify the size of the stack the task uses. The value defaults to 4096 bytes. Take care to set this value to a value that is large enough. If the task uses more than the allotted space it can write into other memory areas with unintended results.

Each DSP/BIOS task has a separate stack. This parameter is not related to **System stack size (MAUs)** in the model Configuration Parameters.

Stack memory segment

Specify where the stack resides in memory by specifying the memory segment. Additional information about DSP/BIOS memory segments also appears in the Target Hardware Resources tab.

Synchronize data transfer of this task with caller task

Specify whether this task should synchronize data transfer with the calling task. Select this option to enable synchronization. Clearing this option enables the **Timer resolution** option.

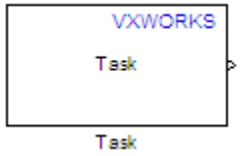
Timer resolution

When you direct the block not to synchronize data with the calling task (by clearing **Synchronize data transfer of this task with caller task**), **Timer resolution** reports the resolution of the timer. **Timer resolution** is a read-only parameter. You cannot change the value.

Introduced in R2011a

VxWorks Task

Spawn task function as separate VxWorks thread

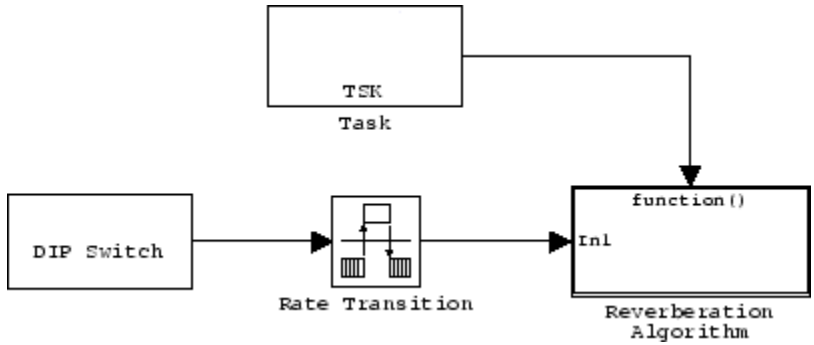


Library

Embedded Coder Support Package for Wind River® VxWorks® RTOS

Description

Use this block to create a task function that spawns as a separate VxWorks thread. The task function runs the code of the downstream function-call subsystem. For example:

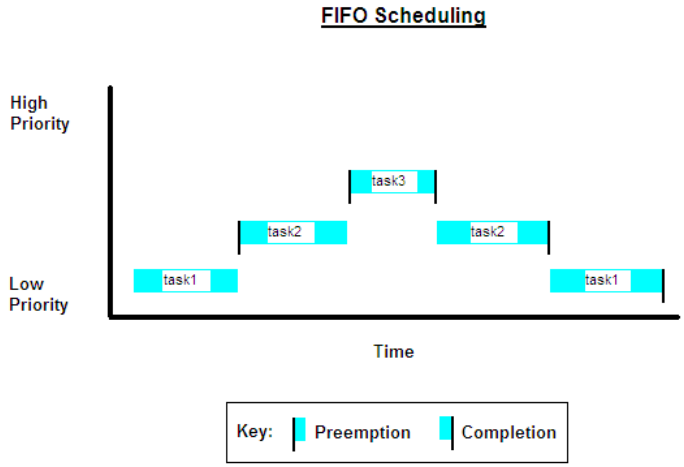


In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_grt.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

The VxWorks Task block uses a First In, First Out (FIFO) scheduling algorithm, which executes real-time processes without time slicing. With FIFO scheduling, a higher-priority

process preempts a lower-priority process. While the higher-priority process runs, the lower-priority process remains at the top of the list for its priority. When the scheduler blocks the higher-priority processes, the lower-priority process resumes.

For example, in the following image, task2 preempts task1. Then, task3 preempts task2. When task3 completes, task2 resumes. When task2 completes, task1 resumes.



Parameters

Task function name (32 characters or less)

Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

Thread priority (0-255)

Set the priority for the thread, from 0 to 255 (low-to-high). Higher-priority tasks can preempt lower-priority tasks.

Introduced in R2011a

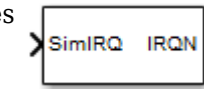
4

Blocks in Simulink Coder—Alphabetical List

Async Interrupt

Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that execute downstream subsystems or Task Sync blocks

Library: Simulink Coder / Asynchronous / Interrupt Templates



Description

For each specified VME interrupt level in the example RTOS (VxWorks), the Async Interrupt block generates an interrupt service routine (ISR) that calls one of the following:

- A function call subsystem
- A Task Sync block
- A Stateflow chart configured for a function call input event

Note You can use the blocks in the `vxlib1` (Simulink Coder) library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Assumptions and Limitations

- The block supports VME interrupts 1 through 7.
- The block uses these RTOS (VxWorks) system calls:

```
sysIntEnable
sysIntDisable
intConnect
intLock
intUnlock
tickGet
```

Performance Considerations

Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. Usually, it is best to keep ISRs as short as possible. Connect only function-call subsystems that contain a few blocks to an Async Interrupt block.

A better solution for large subsystems is using the Task Sync block to synchronize the execution of the function-call subsystem to an RTOS task. Place the Task Sync block between the Async Interrupt block and the function-call subsystem. The Async Interrupt block then uses the Task Sync block as the ISR. The ISR releases a synchronization semaphore (performs a `semGive`) to the task, and returns immediately from interrupt level. The example RTOS (VxWorks) then schedules and runs the task. See the description of the Task Sync block.

Ports

Input

`Input` — Simulated interrupt source

scalar

A simulated interrupt source.

Output Arguments

`Output` — Control signal

scalar

Control signal for a:

- Function-call subsystem
- Task Sync block
- Stateflow chart configured for a function call input event

Parameters

`VME interrupt number(s)` — VME interrupt numbers for the interrupts to be installed
[1 2] (default) | integer array

An array of VME interrupt numbers for the interrupts to be installed. The valid range is `1..7`.

The width of the Async Interrupt block output signal corresponds to the number of VME interrupt numbers specified.

Note A model can contain more than one Async Interrupt block. However, if you use more than one Async Interrupt block, do not duplicate the VME interrupt numbers specified in each block.

`VME interrupt vector offset(s)` — Interrupt vector offset numbers corresponding to the VME interrupt numbers
[192 193] (default) | integer array

An array of unique interrupt vector offset numbers corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field. The Stateflow software passes the offsets to the RTOS (VxWorks) call `intConnect(INUM_TO_IVEC(offset),...)`.

`Simulink task priority(s)` — Priority of downstream blocks
[10 11] (default) | integer array

The Simulink priority of downstream blocks. Each output of the Async Interrupt block drives a downstream block (for example, a function-call subsystem). Specify an array of priorities corresponding to the VME interrupt numbers that you specify for **VME interrupt number(s)**.

The **Simulink task priority** values are required to generate a rate transition code (see "Rate Transitions and Asynchronous Blocks" (Simulink Coder)). Simulink task priority values are also required to maintain absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. The assigned priorities typically are higher than the priorities assigned to periodic tasks.

Note The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

`Preemption flag(s); preemptable-1; non-preemptable-0` — Selects preemption

[0 1] (default) | integer array

Set this option to 1 if an output signal of the Async Interrupt block drives a Task Sync block.

Higher priority interrupts can preempt lower priority interrupts in the example RTOS (VxWorks). To lock out interrupts during the execution of an ISR, set the pre-emption flag to 0. This setting causes generation of `intLock()` and `intUnlock()` calls at the beginning and end of the ISR code. Use interrupt locking carefully, as it increases the interrupt response time of the system for interrupts at the `intLockLevelSet()` level and below. Specify an array of flags corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field.

Note The number of elements in the arrays specifying **VME interrupt vector offset(s)** and **Simulink task priority** must match the number of elements in the **VME interrupt number(s)** array.

`Manage own timer` — Select timer manager

on (default) | off

If selected, the ISR generated by the Async Interrupt block manages its own timer by reading absolute time from the hardware timer. Specify the size of the hardware timer with the **Timer size** option.

`Timer resolution (seconds)` — Resolution of ISR timer

1/60 (default)

The resolution of the ISRs timer. ISRs generated by the Async Interrupt block maintain their own absolute time counters. By default, these timers obtain their values from the RTOS (VxWorks) kernel by using the `tickGet` call. The **Timer resolution** field determines the resolution of these counters. The default resolution is `1/60` second. The `tickGet` resolution for your board support package (BSP) can be different. Determine the `tickGet` resolution for your BSP and enter it in the **Timer resolution** field.

If you are targeting an RTOS other than the example RTOS (VxWorks), replace the `tickGet` call with an equivalent call to the target RTOS. Or, generate code to read the timer register on the target hardware. For more information, see “Timers in Asynchronous Tasks” (Simulink Coder) and “Async Interrupt Block Implementation” (Simulink Coder).

`Timer size` — Number of bits to store the clock tick
`32bits` (default) | `16bits` | `8bits` | `auto`

The number of bits to store the clock tick for a hardware timer. The ISR generated by the Async Interrupt block uses the timer size when you select **Manage own timer**. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the code generator determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. When **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, the code generator uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters” (Simulink Coder). See also “Timers in Asynchronous Tasks” (Simulink Coder).

`Enable simulation input` — Select add simulation input port
on (default) | off

If selected, the Simulink software adds an input port to the Async Interrupt block. This port is for simulation only. Connect one or more simulated interrupt sources to the simulation input.

Note Before generating code, consider removing blocks that drive the simulation input to prevent the blocks from contributing to the generated code. Alternatively, you can use the Environment Controller block, as explained in “Dual-Model Approach: Code Generation” (Simulink Coder). If you use the Environment Controller block, the sample times of driving blocks contribute to the sample times supported in the generated code.

See Also

Task Sync

Topics

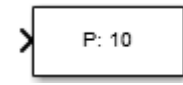
“Asynchronous Events” (Simulink Coder)
“Asynchronous Events” (Simulink Coder)

Introduced in R2006a

Asynchronous Task Specification

Specify priority of asynchronous task represented by referenced model triggered by asynchronous interrupt

Library: Simulink Coder / Asynchronous



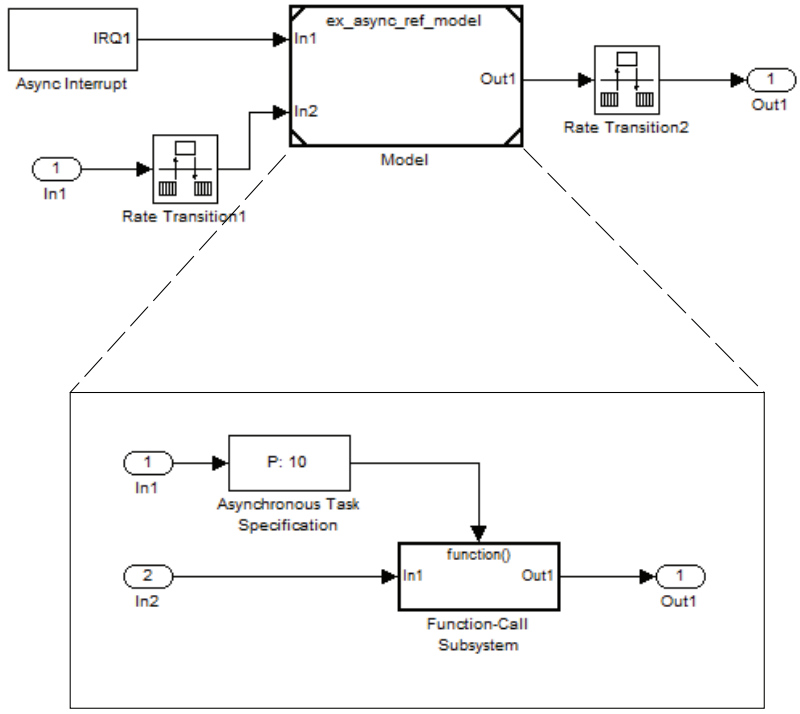
Description

The Asynchronous Task Specification block specifies parameters, such as the task priority, of an asynchronous task represented by a function-call subsystem with a trigger from an asynchronous interrupt. Use this block to control scheduling of function-call subsystems with triggers from asynchronous events. You control the scheduling by assigning a priority to each function-call subsystem within a referenced model.

To use this block, follow the procedure in “Convert an Asynchronous Subsystem into a Model Reference” (Simulink Coder).

Observe in the figure:

- The block must reside in a referenced model between a root-level Inport block and a function-call subsystem. The Asynchronous Task Specification block must immediately follow and connect directly to the Inport block.
- The Inport block must receive an interrupt signal from an Async Interrupt block that is in the parent model.
- The Inport block must be configured to receive and send function-call trigger signals.



Ports

Input

`Port_1` — Interrupt input signal

scalar

Interrupt input signal received from a root-level Inport block.

Output

`Port_1` — Interrupt signal with priority

scalar

Interrupt signal with specified task priority that triggers a function-call subsystem.

Parameters

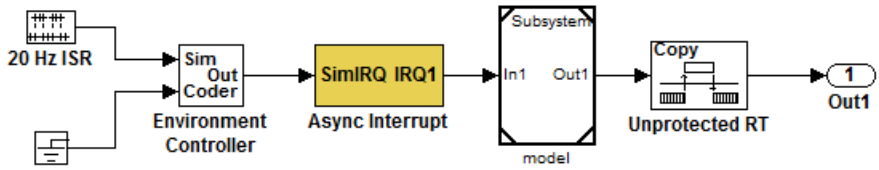
`Task priority` — Priority of asynchronous task that calls function-call subsystem

`10` (default)

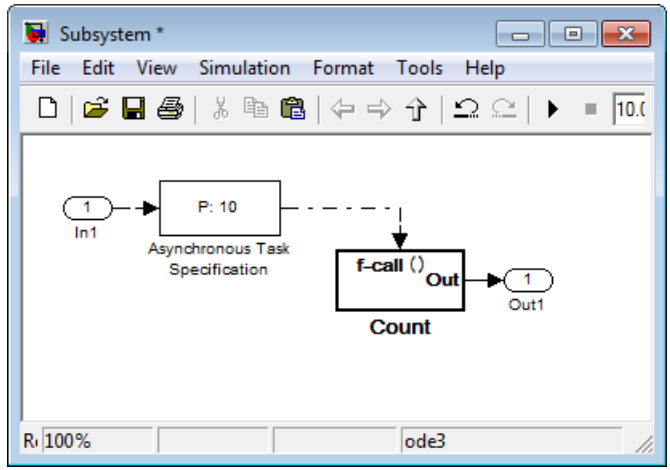
Specify an integer or [] as the priority of the asynchronous task that calls the connected function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

- If you specify an integer, it must match the priority value of the interrupt signal initiator in the parent model.
- If you specify [], the priority does not have to match the priority of the interrupt signal initiator in the top model. The rate transition algorithm is conservative (not optimized). The priority is unknown but static.

Consider the following model.



The referenced model has the following content.



If the **Task priority** parameter is set to 10, the Async Interrupt block in the parent model must also have a priority of 10. If the parameter is set to [], the priority of the Async Interrupt block can be a value other than 10.

See Also

Blocks
Function-Call Subsystem | Inport

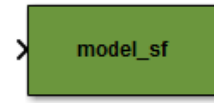
Topics

“Asynchronous Events” (Simulink Coder)
“Spawn and Synchronize Execution of RTOS Task” (Simulink Coder)
“Pass Asynchronous Events in RTOS as Input To a Referenced Model” (Simulink Coder)
“Convert an Asynchronous Subsystem into a Model Reference” (Simulink Coder)
“Rate Transitions and Asynchronous Blocks” (Simulink Coder)
“Asynchronous Support” (Simulink Coder)
“Asynchronous Events” (Simulink Coder)
“Model References” (Simulink)

Introduced in R2011a

Generated S-Function

Represent model or subsystem as generated S-function code
Library: Simulink Coder / S-Function Target



Description

An instance of the Generated S-Function block represents code that the code generator produces from its S-function system target file for a model or subsystem. For example, you extract a subsystem from a model and build a Generated S-Function block from it by using the S-function target. This mechanism can be useful for:

- Converting models and subsystems to application components
- Reusing models and subsystems
- Optimizing simulation—often, an S-function simulates more efficiently than the original model

For details on how to create a Generated S-Function block from a subsystem, see “Create S-Function Blocks from a Subsystem” (Simulink Coder).

Requirements

- The S-Function block must perform identically to the model or subsystem from which it was generated.
- Before creating the block, explicitly specify Inport block signal attributes, such as signal widths or sample times. The sole exception to this rule concerns sample times, as described in “Sample Time Propagation in Generated S-Functions” (Simulink Coder).
- Set the solver parameters of the Generated S-Function block to be the same as the parameters of the original model or subsystem. The generated S-function code operates identically to the original subsystem (for an exception to this rule, see “Choose a Solver Type” (Simulink Coder)).

Ports

Input

`Input` — S-function input
varies

See requirements.

Output Arguments

`Output` — S-function output
varies

See requirements.

Parameters

`Generated S-function name (model_sf)` — Name of S-function
model_sf (default) | character vector

The name of the generated S-function. The code generator derives the name by appending `_sf` to the name of the model or subsystem from which the block is generated.

`Show module list` — Select display module list
off (default) | on

If selected, displays modules generated for the S-function.

See Also

Topics

“Generate S-Function from Subsystem” (Simulink Coder)
“Create S-Function Blocks from a Subsystem” (Simulink Coder)

Introduced in R2011b

Model Header

Specify external header code

Description

For a model that includes the Model Header block, the code generator adds external code that you specify to the header file (*`model`*`.h`) that it generates. You can specify code for the code generator to add near the top and bottom of the header file.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`Top of Model Header` — Code to add near top of generated header file

Specify code that you want the code generator to add near the top of the header file for the model. The code generator places the code in the section labeled `user code (top of header file)`.

`Bottom of Model Header` — Code to add at bottom of generated header file

Specify code that you want the code generator to add at the bottom of the header file for the model. The code generator places the code in the section labeled `user code (bottom of header file)`.

See Also

Model Source | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

Model Source

Specify external source code

Description

For a model that includes the Model Source block, the code generator adds external code that you specify to the source file (*`model`*`.c` or *`model`*`.cpp`) that it generates. You can specify code for the code generator to add near the top and bottom of the source file.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`Top of Model Header` — Code to add near top of generated source file

Specify code that you want the code generator to add near the top of the source file for the model. The code generator places the code in the section labeled `user code (top of source file)`.

`Bottom of Model Header` — Code to add at bottom of generated source file

Specify code that you want the code generator to add at the bottom of the source file for the model. The code generator places the code in the section labeled `user code (bottom of source file)`.

Example

See “Add External Code to Generated Start Function” (Simulink Coder).

See Also

Model Header | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

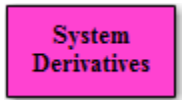
Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Derivatives

Specify external system derivative code



Description

For a model or nonvirtual subsystem that includes the System Derivatives block and a block that computes continuous states, the code generator adds external code, which you specify, to the `SystemDerivatives` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Derivatives Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemDerivatives` function for the model or subsystem.

`System Derivatives Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemDerivatives` function for the model or subsystem.

`System Derivatives Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemDerivatives` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Disable | System Enable | System Outputs | System Start | System Terminate | System Update

Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Disable

Specify external system disable code

Description

For a model or nonvirtual subsystem that includes the System Disable block and a block that uses a `SystemDisable` function, the code generator adds external code, which you specify, to the `SystemDisable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Disable Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemDisable` function for the model or subsystem.

`System Disable Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemDisable` function for the model or subsystem.

`System Disable Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemDisable` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Derivatives | System Enable | System Outputs | System Start | System Terminate | System Update

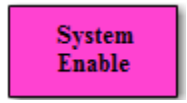
Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Enable

Specify external system enable code



Description

For a model or nonvirtual subsystem that includes the System Enable block and a block that uses a `SystemEnable` function, the code generator adds external code, which you specify, to the `SystemEnable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Enable Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemEnable` function for the model or subsystem.

`System Enable Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemEnable` function for the model or subsystem.

`System Enable Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemEnable` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

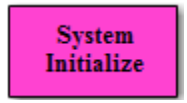
Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Initialize

Specify external system initialization code



Description

For a model or nonvirtual subsystem that includes the System Initialize block and a block that uses a `SystemInitialize` function, the code generator adds external code, which you specify, to the `SystemInitialize` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Initialize Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemInitialize` function for the model or subsystem.

`System Initialize Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemInitialize` function for the model or subsystem.

`System Initialize Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemInitialize` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

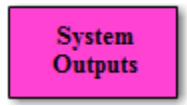
Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Outputs

Specify external system outputs code



Description

For a model or nonvirtual subsystem that includes the System Outputs block and a block that uses a `SystemOutputs` function, the code generator adds external code, which you specify, to the `SystemOutputs` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Outputs Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemOutputs` function for the model or subsystem.

`System Outputs Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemOutputs` function for the model or subsystem.

`System Outputs Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemOutputs` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Initialize | System Start | System Terminate | System Update

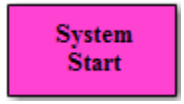
Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Start

Specify external system startup code



Description

For a model or nonvirtual subsystem that includes the System Start block and a block that uses a `SystemStart` function, the code generator adds external code, which you specify, to the `SystemStart` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Start Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemStart` function for the model or subsystem.

`System Start Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemStart` function for the model or subsystem.

`System Start Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemStart` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Terminate | System Derivatives | System Disable | System Initialize | System Outputs | System Update

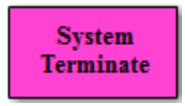
Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Terminate

Specify external system termination code



Description

For a model or nonvirtual subsystem that includes the System Terminate block and a block that uses a `SystemTerminate` function, the code generator adds external code, which you specify, to the `SystemTerminate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Terminate Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemTerminate` function for the model or subsystem.

`System Disable Terminate Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemTerminate` function for the model or subsystem.

`System Disable Terminate Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemTerminate` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Update

Topics

“Place External C/C++ Code in Generated Code”

Introduced in R2006a

System Update

Specify external system update code

Description

For a model or nonvirtual subsystem that includes the System Update block and a block that uses a `SystemUpdate` function, the code generator adds external code, which you specify, to the `SystemUpdate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Update Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemUpdate` function for the model or subsystem.

`System Update Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemUpdate` function for the model or subsystem.

`System Update Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemUpdate` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Terminate

Topics

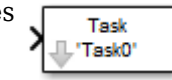
“Place External C/C++ Code in Generated Code”

Introduced in R2006a

Task Sync

Run code of downstream function-call subsystem or Stateflow chart by spawning an example RTOS (VxWorks) task

Library: Simulink Coder / Asynchronous / Interrupt Templates



Description

The Task Sync block spawns an example RTOS (VxWorks) task that calls a function-call subsystem or Stateflow chart. Typically, you place the Task Sync block between an Async Interrupt block and a function-call subsystem block or Stateflow chart. Alternatively, you could connect the Task Sync block to the output port of a Stateflow diagram that has an event, `Output to Simulink`, configured as a function call.

The Task Sync block:

- Uses the RTOS (VxWorks) system call `taskSpawn` to spawn an independent task. When the task is activated, it calls the downstream function-call subsystem code or Stateflow chart. The block calls `taskDelete` to delete the task during model termination.
- Creates a semaphore to synchronize the connected subsystem with execution of the block.
- Wraps the spawned task in an infinite `for` loop. In the loop, the spawned task listens for the semaphore by using `semTake`. The first call to `semTake` specifies `NO_WAIT`. This setting lets the task determine whether a second `semGive` has occurred before the completion of the function-call subsystem or chart. This sequence indicates that the interrupt rate is too fast or the task priority is too low.
- Generates synchronization code (for example, `semGive()`). This code lets the spawned task run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. The connection between the Async Interrupt and Task Sync blocks accomplishes this operation and triggers execution of the Task Sync block within an ISR.
- Supplies absolute time if blocks in the downstream algorithmic code require it. The time comes from the timer maintained by the Async Interrupt block or comes from an independent timer maintained by the task associated with the Task Sync block.

When you design your application, consider when timer and signal input values could be taken for the downstream function-call subsystem that is connected to the Task Sync block. By default, the time and input data are read when the RTOS (VxWorks) activates the task. For this case, the data (input and time) are synchronized to the task itself. If you select the **Synchronize the data transfer of this task with the caller task** option and the Task Sync block driver is an Async Interrupt block, the time and input data are read when the interrupt occurs (that is, within the ISR). For this case, data is synchronized with the caller of the Task Sync block.

Note You can use the blocks in the `vxlib1` (Simulink Coder) library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Ports

Input

`Input` — Call from interrupt block
call

A call from an Async Interrupt block.

Output Arguments

`Output` — Call to function-call subsystem
call

A call to a function-call subsystem.

Parameters

`Task name (10 characters or less)` — Task function name
`Task0` (default) | character vector

The first argument passed to the `taskSpawn` system call in the RTOS. The RTOS (VxWorks) uses this name as the task function name. This name also serves as a

debugging aid. Routines use the task name to identify the task from which they are called.

`Simulink task priority (0–255)` — RTOS task priority
50 (default) | integer

The RTOS task priority assigned to the function-call subsystem task when spawned. RTOS (VxWorks) priorities range from 0 to 255, with 0 representing the highest priority.

Note The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

`Stack size (bytes)` — Maximum size for stack of the task
1024 (default) | integer

Maximum size to which the stack of the task can grow. The stack size is allocated when the RTOS (VxWorks) spawns the task. Choose a stack size based on the number of local variables in the task. Determine the size by examining the generated code for the task (and functions that are called from the generated code).

`Synchronize the data transfer of this task with the caller task` — Select synchronization
off (default) | on

If not selected (the default),

- The block maintains a timer that provides absolute time values required by the computations of downstream blocks. The timer is independent of the timer maintained by the Async Interrupt block that calls the Task Sync block.
- A **Timer resolution** option appears.
- The **Timer size** option specifies the word size of the time counter.

If selected,

- The block does not maintain an independent timer and does not display the **Timer resolution** field.
- Downstream blocks that require timers use the timer maintained by the Async Interrupt block that calls the Task Sync block (see “Timers in Asynchronous Tasks”

(Simulink Coder)). The timer value is read at the time the asynchronous interrupt is serviced. Data transfers to blocks called by the Task Sync block execute within the task associated with the Async Interrupt block. Therefore, data transfers are synchronized with the caller.

`Timer resolution (seconds)` — Resolution for timer of the block
1/60 (default)

The resolution of the timer of the block in seconds. This option appears only if **Synchronize the data transfer of this task with the caller task** is not selected. By default, the block gets the timer value by calling the `tickGet` function in the RTOS (VxWorks). The default resolution is 1/60 second.

`Timer size` — Number of bits to store clock tick
`32bits` (default) | `16bits` | `8bits` | `auto`

The number of bits to store the clock tick for a hardware timer. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the code generator determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. When **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, it uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters” (Simulink Coder). See also “Timers in Asynchronous Tasks” (Simulink Coder).

See Also
Async Interrupt

Topics
“Asynchronous Events” (Simulink Coder)

Introduced in R2006a

5

Embedded Coder Parameters: Advanced Parameters

Create block

Description

Generate a SIL or PIL block

Category: Code Generation > Verification

Settings

Default: `None`

`None`

SIL or PIL block not generated.

`SIL`

Generate a SIL block that represents a top-model or subsystem.

If you select this option, the software creates and opens an untitled model with a SIL block. The SIL block contains an S-function, through which the software runs compiled object code on the host computer. With this block, you can verify the behavior of source code generated from top-model or subsystem components.

If the subsystem is an export-function subsystem, the software creates a Model block with **Simulation mode** set to `Software-in-the-loop (SIL)`.

`PIL`

Generate a PIL block that represents a top-model or subsystem.

If you select this option, the software creates and opens an untitled model with a PIL block. The PIL block contains an S-function, through which the software runs cross-compiled object code on a target processor or instruction set simulator. With this block, you can verify the behavior of object code generated from top-model or subsystem components.

If the subsystem is an export-function subsystem, the software creates a Model block with **Simulation mode** set to `Processor-in-the-loop (PIL)`.

To control the way code compiles and executes in the target environment, use Target Connectivity API.

Command-Line Information

Parameter: `CreateSILPILBlock`
Type: character vector
Value: `'None'` | `'SIL'` | `'PIL'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Simulation with Blocks From Subsystems”
- “Generate Code for Export-Function Subsystems”
- “SIL and PIL Simulations”
- “Generate Code for Export-Function Subsystems”

Existing shared code

Description

Specify folder that contains existing shared code

Category: Code Generation Advanced Parameters

Settings

Default: none

Path to folder that contains existing shared code. Specify the absolute path or a path relative to the Simulink preference **Code generation folder** (`CodeGenFolder`). The model build process uses the code in this folder instead of locally generated shared utility code.

Command-Line Information

Parameter: `ExistingSharedCode`
Type: character vector
Value: valid MATLAB variable name
Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact
Safety precaution	No impact

See Also

`sharedCodeUpdate`

Related Examples

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

Use only existing shared code

Description

Check whether build process requires shared code that is not present in the existing shared code folder.

Category: Diagnostics

Settings

Default: `none`

`none`

Simulink software takes no action.

`warning`

Simulink software displays a warning.

`error`

Simulink software terminates the simulation and displays an error message.

Dependency

You can specify this parameter only if you specify a folder in the **Existing shared code** field. Otherwise the field appears dimmed.

Command-Line Information

Parameter: `UseOnlyExistingSharedCode`
Type: character vector
Value: `'none'` | `'warning'` | `'error'`
Default: `'none'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Cross-Release Shared Utility Code Reuse”
- “Cross-Release Code Integration”

Use Embedded Coder Features

Description

Enable “Embedded Coder” features for models deployed to “Simulink Supported Hardware” (Simulink).

Note If you enable this parameter in a model where Embedded Coder is not installed or available in the environment, a question dialog box prompts you to update the model to build without Embedded Coder features.

Category: Hardware Implementation

Settings

Default: `On`

 On

Enable advanced Embedded Coder configuration parameters.

Note Enabling `Use Embedded Coder Features` also enables the “Use Simulink Coder Features” (Simulink Coder) parameter.

 Off

Disable advanced Embedded Coder configuration parameters.

Dependencies

This parameter requires an Embedded Coder license.

Command-Line Information

Parameter: `UseEmbeddedCoderFeatures`
Value: `'on'` or `'off'`

Default: `'on'`

See Also

Related Examples

- “Hardware Implementation Pane” (Simulink)

Remove reset function

Description

Remove unreachable (dead-code) instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies. If you enable this parameter, Simulink checks that live code will be removed and errors if it finds such code.

Category: Code Generation > Interface

Settings

Default: `On`

 On

Remove unreachable instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies.

 Off

Generate code without removing unreachable instances of the `reset` function.

Dependencies

This parameter requires an Embedded Coder license.

To set the **Remove reset function** parameter, set **Configuration Parameters > Code Generation > System target file** parameter to an ERT-based system target file, such as `ert.tlc`.

Command-Line Information

Parameter: `RemoveResetFunc`
Value: `'on'` or `'off'`
Default: `'on'`

See Also

Related Examples

- “Remove disable function” on page 5-12
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)

Remove disable function

Description

Remove unreachable (dead-code) instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies. If you enable this parameter, Simulink checks that live code will be removed and errors if it finds such code.

Category: Code Generation > Interface

Settings

Default: `Off`

 On

Remove unreachable instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies.

 Off

Generate code without removing unreachable instances of the `disable` function.

Dependencies

This parameter requires an Embedded Coder license.

To set the **Remove disable function** parameter, set **Configuration Parameters > Code Generation > System target file** parameter to an ERT-based system target file, such as `ert.tlc`.

Command-Line Information

Parameter: `RemoveDisableFunc`
Value: `'on'` or `'off'`
Default: `'off'`

See Also

Related Examples

- “Remove reset function” on page 5-10
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)

6

Code Generation Parameters: AUTOSAR

Model Configuration Parameters: Code Generation AUTOSAR

The **Code Generation > AUTOSAR Code Generation Options** category includes parameters for controlling AUTOSAR code generation. On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > AUTOSAR Code Generation Options** pane.

Parameter	Description
"Generate XML file for schema version" on page 6-4	Select the AUTOSAR schema version to use when generating XML files.
"Maximum SHORT-NAME length" on page 6-6	Specify maximum length for `SHORT-NAME` XML elements.
"Use AUTOSAR compiler abstraction macros" on page 6-7	Specify use of AUTOSAR macros to abstract compiler directives.
"Support root-level matrix I/O using one-dimensional arrays" on page 6-9	Allow root-level matrix I/O.

See Also

More About

- "Code Generation" (AUTOSAR Blockset)
- "Model Configuration"

Code Generation: AUTOSAR Code Generation Options Tab Overview

Parameters for controlling AUTOSAR code generation options.

Configuration

This pane appears only if you specify the `autosar.tlc` system target file.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

Tip

From the Simulink **Code** menu, select **C/C++ Code** > **Configure Model as AUTOSAR Component** to open a dialog box where you can configure other AUTOSAR options.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Code Generation” (AUTOSAR Blockset)

Generate XML file for schema version

Description

Select the AUTOSAR schema version to use when generating XML files.

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: `4.3`

`4.3`
: Use schema version 4.3 (revision 4.3.1)

`4.2`
: Use schema version 4.2 (revision 4.2.2)

`4.1`
: Use schema version 4.1 (revision 4.1.3)

`4.0`
: Use schema version 4.0 (revision 4.0.3)

`3.2`
: Use schema version 3.2 (revision 3.2.2)

`3.1`
: Use schema version 3.1 (revision 3.1.4)

`3.0`
: Use schema version 3.0 (revision 3.0.2)

`2.1`
: Use schema version 2.1 (XSD rev 0017)

Tip

- Selecting the AUTOSAR target for your model for the first time sets the schema version parameter to the default value, `4.3`.

- When you import `arxml` code into Simulink, the `arxml` importer detects the schema version and sets the schema version parameter in the model. For a list of AUTOSAR schema revisions supported for `arxml` import, see “Select an AUTOSAR Schema” (AUTOSAR Blockset).
- Must be set to the same value for top and referenced models.
- To configure other AUTOSAR XML options, from the Simulink **Code** menu, select **C/C++ Code** > **Configure AUTOSAR Dictionary**. Select **XML Options**.

Command-Line Information

Parameter: `AutosarSchemaVersion`
Type: character vector
Value: `'4.3'` | `'4.2'` | `'4.1'` | `'4.0'` | `'3.2'` | `'3.1'` | `'3.0'` | `'2.1'`
Default: `'4.3'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Code Generation” (AUTOSAR Blockset)

Maximum SHORT-NAME length

Description

Specify maximum length for `SHORT-NAME` XML elements

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: 128

The AUTOSAR standard specifies that the length of `SHORT-NAME` XML elements cannot be greater than 128 characters, for schema version 4.x or later, or 32 characters, for earlier schema versions. Use this parameter to specify a maximum length for SHORT-NAME elements exported by the code generator, up to 128 characters.

Tip

Must be set to the same value for top and referenced models.

Command-Line Information

Parameter: `AutosarMaxShortNameLength`
Type: integer
Value: an integer less or equal to 128
Default: 128

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Specify Maximum SHORT-NAME Length” (AUTOSAR Blockset)

Use AUTOSAR compiler abstraction macros

Description

Specify use of AUTOSAR macros to abstract compiler directives

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: Off

 On

Software generates code with C macros that are abstracted compiler directives (near/far memory calls)

 Off

Software generates code that does *not* contain AUTOSAR compiler abstraction macros.

Tip

Must be the same for top and referenced models.

Command-Line Information

Parameter: `AutosarCompilerAbstraction`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

- “Configure AUTOSAR Compiler Abstraction Macros” (AUTOSAR Blockset)

Support root-level matrix I/O using one-dimensional arrays

Description

Allow root-level matrix I/O

Category: Code Generation > AUTOSAR Code Generation Options

Settings

Default: Off

 On

Software supports matrix I/O at the root-level by generating code that implements matrices as one-dimensional arrays.

 Off

Software does not allow matrix I/O at the root-level. If you try to build a model that has matrix I/O at the root-level, the software produces an error.

Tip

Must be the same for top and referenced models.

Command-Line Information

Parameter: `AutosarMatrixIOAsArray`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2
- “Root-Level Matrix I/O” (AUTOSAR Blockset)

Code Generation Parameters: Code Placement

Model Configuration Parameters: Code Generation Code Placement

The **Code Generation > Code Placement** category includes parameters for configuring the appearance of the generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > Code Placement** pane.

Parameter	Description
"Data definition" on page 7-5	Specify where to place definitions of global variables.
"Data definition filename" on page 7-7	Specify the name of the file that is to contain data definitions.
"Data declaration" on page 7-9	Specify where `extern`, `typedef`, and `#define` statements are to be declared.
"Data declaration filename" on page 7-11	Specify the name of the file that is to contain data declarations.
"#include file delimiter" on page 7-15	Specify the type of `#include` file delimiter to use in generated code.
"Use owner from data object for data definition placement" on page 7-13	Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.
"Signal display level" on page 7-17	Specify the persistence level for MPT signal data objects.
"Parameter tune level" on page 7-19	Specify the persistence level for MPT parameter data objects.
"File packaging format" on page 7-21	Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files.
"Header files" on page 7-24	Specify customized name for generated header files.
"Source files" on page 7-26	Specify customized name for generated source files.

Parameter	Description
“Data files” on page 7-28	Specify customized name for generated data files.
“Rate Transition block code” on page 7-30	Specify the format for Rate Transition block code and data.

See Also

More About

- “Model Configuration”

Code Generation: Code Placement Tab Overview

Specify the data placement in the generated code.

Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2

Data definition

Description

Specify where to place definitions of global variables.

Category: Code Generation > Code Placement

Settings

Default: `Auto`

`Auto`

Lets the code generator determine where the definitions should be located.

`Data defined in source file`

Places 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 the Simulink model.

`Data defined in a single separate source file`

Places 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

- This parameter applies to data with custom storage classes only.
- This parameter enables **Data definition filename**.

Command-Line Information

Parameter: `GlobalDataDefinition`
Type: character vector
Value: `'Auto'` | `'InSourceFile'` | `'InSeparateSourceFile'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Control Placement of Global Data Definitions and Declarations in Generated Files”

Data definition filename

Description

Specify the name of the file that is to contain data definitions.

Category: Code Generation > Code Placement

Settings

Default: `global.c`

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 **Code Generation** pane: **Templates** tab.

You do not need to specify an extension for the file name. If you want to specify an extension, you must use a `.c` extension. In either case:

- If you select C as the target language, the code generator creates a file with a `.c` extension.
- If you select C++ as the target language, the code generator creates a file with a `.cpp` extension.

Limitation

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

Dependency

This parameter is enabled by **Data definition**.

Command-Line Information

Parameter: `DataDefinitionFile`
Type: character vector
Value: a valid file
Default: `'global.c'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid file
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting and Defining Templates
- Custom File Processing

Data declaration

Description

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

Category: Code Generation > Code Placement

Settings

Default: `Auto`

`Auto`

Lets the code generator determine where the declarations should be located.

`Data declared in source file`

Places 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 the Simulink model.

`Data defined in a single separate source file`

Places 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 **Code Generation** pane: **Templates** tab.

Dependencies

- This parameter applies to data with custom storage classes only.
- This parameter enables **Data declaration filename**.

Command-Line Information

Parameter: `GlobalDataReference`
Type: character vector
Value: `'Auto'` | `'InSourceFile'` | `'InSeparateHeaderFile'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Control Placement of Global Data Definitions and Declarations in Generated Files”

Data declaration filename

Description

Specify the name of the file that is to contain data declarations.

Category: Code Generation > Code Placement

Settings

Default: `global.h`

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 **Code Generation** pane: **Templates** tab.

Limitation

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

Dependency

This parameter is enabled by **Data declaration**.

Command-Line Information

Parameter: `DataReferenceFile`
Type: character vector
Value: a valid file
Default: `'global.h'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid file

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting and Defining Templates
- Custom File Processing

Use owner from data object for data definition placement

Description

Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.

Category: Code Generation > Code Placement

Settings

Default: off

 On

Uses the ownership setting of the data object for data definition.

 Off

Ignores the ownership setting of the data object for data definition.

Command-Line Information

Parameter: `EnableDataOwnership`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2

#include file delimiter

Description

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

Category: Code Generation > Code Placement

Settings

Default: `Auto`

`Auto`

Lets the code generator choose the `#include` file delimiter

`#include “header.h”`

Uses double quote (" ") characters to delimit file names in `#include` statements.

`#include <header.h>`

Uses 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 parameter.

Command-Line Information

Parameter: `IncludeFileDelimiter`
Type: character vector
Value: `'Auto'` | `'UseQuote'` | `'UseBracket'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2

Signal display level

Description

Specify the persistence level for MPT signal data objects.

Category: Code Generation > Code Placement

Settings

Default: 10

Specify an integer value indicating the persistence level for 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 that you can specify for a specific MPT signal data object in the Model Explorer signal properties dialog.

Dependency

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

Command-Line Information

Parameter: `SignalDisplayLevel`
Type: integer
Value: a valid integer
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid integer

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting Persistence Level for Signals and Parameters

Parameter tune level

Description

Specify the persistence level for MPT parameter data objects.

Category: Code Generation > Code Placement

Settings

Default: 10

Specify an integer value indicating the persistence level for 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 that can specify for a specific MPT parameter data object in the Model Explorer parameter properties dialog.

Dependency

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

Command-Line Information

Parameter: `ParamTuneLevel`
Type: integer
Value: a valid integer
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid integer

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- Selecting Persistence Level for Signals and Parameters

File packaging format

Description

Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files. You can specify a different file packaging format for each referenced model.

Category: Code Generation > Code Placement

Settings

Default: `Modular`

`Modular`

- Outputs *`model`*`_data.c`, *`model`*`_private.h`, and *`model`*`_types.h`, in addition to generating *`model`*`.c` and *`model`*`.h`. For the contents of these files, see the table in "Generated Code Modules".
- Supports generating separate source files for subsystems. For more information on generating code for subsystems, see "Control Generation of Functions for Subsystems" (Simulink Coder).
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, some utility files are in the build directory. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.

`Compact (with separate data file)`

- Conditionally outputs *`model`*`_data.c`, in addition to generating *`model`*`.c` and *`model`*`.h`.
- If you specify **Shared code placement** as `Auto` on the **Code Generation** > **Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *`model`*`.c`. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

`Compact`

- The contents of *`model`*`_data.c` are in *`model`*`.c`.
- The contents of *`model`*`_private.h` and *`model`*`_types.h` are in *`model`*`.h` or *`model`*`.c`.
- If you specify **Shared code placement** as `Auto` on the **Code Generation > Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *`model`*`.c`. If you specify **Shared code placement** as `Shared location`, separate files are generated for utility code in a shared location.
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

Command-Line Information

Parameter: `ERTFilePackagingFormat`
Type: character vector
Value: `'Modular'` | `'CompactWithDataFile'` | `'Compact'`
Default: `'Modular'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated Code Modules”
- “Manage File Packaging of Generated Code Modules”

- “Customize Post-Code-Generation Build Processing” (Simulink Coder)
- “Generate Shared Utility Code” (Simulink Coder)

Header files

Description

Specify customized name for generated header files.

Category: Code Generation > Code Placement

Settings

Default: `$R$E`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of custom user text and these format tokens.

Token	Description
`$E`	Insert the file type. `$E` represents these instances of file types: • `capi` • `capi_host` • `dt` • `testinterface` • `private` • `types` Required.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** (Simulink Coder) parameter.

Custom naming is supported only for `.h` and `.hpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `ERTHeaderFileRootName`
Type: character vector
Value: Valid combination of tokens and custom text
Default: `$R$E`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

Source files

Description

Specify customized name for generated source files.

Category: Code Generation > Code Placement

Settings

Default: `$R$E`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of custom user text and these format tokens:

Token	Description
`$E`	Insert the file type. `$E` represents these instances of file types: • `capi` • `capi_host` • `dt` • `testinterface` • `private` • `types` Required.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** (Simulink Coder) parameter.

Custom naming is supported only for `.c` and `.cpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `ERTSourceFileRootName`
Type: character vector
Value: Valid combination of tokens and custom text
Default: `$R$E`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

Data files

Description

Specify customized name for generated data files.

Category: Code Generation > Code Placement

Settings

Default: `$R_data`

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated identifier. The macro can include a combination of custom user text and these format tokens:

Token	Description
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. To specify this text, use the **Custom token text** (Simulink Coder) parameter.

Custom naming is supported only for `.c` and `.cpp` files. When you have model hierarchy, custom naming is applicable to only the root model.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- `Compact (with separate data file)` option for **File packaging format** on page 7-21 enables this parameter.

Command-Line Information

Parameter: `ERTDataFileRootName`
Type: character vector
Value: Valid combination of tokens and custom text
Default: `$R_data`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Placement” on page 7-2
- “Customize Generated File Names”
- “Identifier Format Control”

Rate Transition block code

Description

Specify the format for Rate Transition block code and data. Inline the code with the model code or create separate functions that the model code calls with state data in a dedicated structure.

Category: Code Generation > Code Placement

Settings

Default: `Inline`

`Inline`

Inline Rate Transition block code with model code. Declare Rate Transition block state data in global block state structure.

`Function`

Separate Rate Transition block code and data from the model code and data. The generated code contains separate `get` and `set` functions that the *`model`*`_step` functions call and a dedicated structure for state data. The generated code also contains separate start and initialize functions that the *`model`*`_initialize` function calls.

Dependencies

- This parameter requires an Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `RateTransitionBlockCode`
Value: `'Inline'|'Function'|`
Default: `'Inline'`

Recommended Settings

Application	Setting
Debugging	Function
Traceability	Function
Efficiency	Inline
Safety precaution	No impact

Note

- The code generator does not separate code and data for Rate Transition blocks that have variable-size signals or are inside a For Each Subsystem block.
- In the Rate Transition block parameters dialog box, you must select the **Ensure data integrity during data transfer** parameter. If you do not select this parameter, the model produces an error during code generation.
- In Configuration Parameters dialog box, the **Multitask rate transition** parameter must be set to `error`. If this parameter is not set to `error`, Embedded Coder disables the **Rate Transition block code** parameter and the code generator inlines Rate Transition block code.

See Also

Related Examples

- “Time-Based Scheduling”

Code Generation Parameters: Code Style

Model Configuration Parameters: Code Generation Code Style

The **Code Generation > Code Style** category includes parameters for configuring the appearance of the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab on the **Code Generation > Code Style** pane.

Parameter	Description
"Parentheses level" on page 8-5	Specify parenthesization style for generated code.
"Preserve operand order in expression" on page 8-7	Specify whether to preserve order of operands in expressions.
"Preserve condition expression in if statement" on page 8-9	Specify whether to preserve empty primary condition expressions in `if` statements.
"Convert if-elseif-else patterns to switch-case statements" on page 8-11	Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements.
"Preserve extern keyword in function declarations" on page 8-13	Specify whether to include the `extern` keyword in function declarations in the generated code.
"Preserve static keyword in function declarations" on page 8-15	Specify whether to include the `static` keyword in function declarations in the generated code.
"Suppress generation of default cases for Stateflow switch statements if unreachable" on page 8-17	Specify whether to generate default cases for switch-case statements in the code for Stateflow charts.
"Replace multiplications by powers of two with signed bitwise shifts" on page 8-19	Specify whether to replace multiplications by powers of two with signed bitwise shifts.
"Allow right shifts on signed integers" on page 8-21	Specify whether to allow signed right bitwise shifts in the generated C/C++ code.

Parameter	Description
“Casting modes” on page 8-23	Specify how the code generator casts data types for variables.
“Indent style” on page 8-25	Specify style for the placement of braces in generated code.
“Indent size” on page 8-27	Specify indent size for generated code.
“Newline style” on page 8-29	Specify the newline style for generated code.
“Maximum line width” on page 8-31	Specify the maximum line width for wrapping the generated code.

See Also

More About

- “Code Appearance”
- “Model Configuration”

Code Generation: Code Style Tab Overview

Control optimizations for readability in generated code.

Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Code Style”

Parentheses level

Description

Specify parenthesization style for generated code.

Category: Code Generation > Code Style

Settings

Default: `Nominal (Optimize for readability)`

Minimum (Rely on C/C++ operators for precedence)

Inserts parentheses only where required by ANSI[1] C or C++, or to override default precedence. For example:

```
Out = In2 - In1 > 1.0 && In2 > 2.0;
```

If you generate C/C++ code using the minimum level, for certain settings in some compilers, you can receive compiler warnings. To eliminate these warnings, try the nominal level.

Nominal (Optimize for readability)

Inserts parentheses in a way that compromises between readability and visual complexity. For example:

```
Out = ((In2 - In1 > 1.0) && (In2 > 2.0));
```

Maximum (Specify precedence with parentheses)

Includes parentheses to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA®[2] requirements. For example:

```
 Out = (((In2 - In1) > 1.0) && (In2 > 2.0));
```

Command-Line Information

Parameter: `ParenthesesLevel`
Type: character vector

1. ANSI is a registered trademark of the American National Standards Institute, Inc.
2. MISRA is a registered trademarks of MIRA Ltd, held on behalf of the MISRA Consortium.

Value: `'Minimum'` | `'Nominal'` | `'Maximum'`
Default: `'Nominal'`

Recommended Settings

Application	Setting
Debugging	`Nominal (Optimized for readability)`
Traceability	`Nominal (Optimized for readability)`
Efficiency	`Minimum (Rely on C/C++ operators for precedence)`
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- Control Parentheses in Generated Code

Preserve operand order in expression

Description

Specify whether to preserve order of operands in expressions.

Category: Code Generation > Code Style

Settings

Default: off

 On

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

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

 Off

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

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

Command-Line Information

Parameter: `PreserveExpressionOrder`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Optimize Code by Reordering Commutable Operands”

Preserve condition expression in if statement

Description

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

Category: Code Generation > Code Style

Settings

Default: off

 On

Preserves 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
```

 Off

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

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

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

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

Command-Line Information

Parameter: `PreserveIfCondition`
Type: character vector

Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Optimize Generated Code by Consolidating Redundant If-Else Statements”

Convert if-elseif-else patterns to switch-case statements

Description

Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements.

This readability optimization works on a per-model basis and applies only to:

- Flow charts in Stateflow charts
- MATLAB functions in Stateflow charts
- MATLAB Function blocks in that model

Category: Code Generation > Code Style

Settings

Default: on

 On

Generate code for `if-elseif-else` decision logic as `switch-case` statements.

For example, assume that you have the following logic pattern:

```
if (x == 1) {
       y = 1;
} else if (x == 2) {
       y = 2;
} else if (x == 3) {
       y = 3;
} else {
       y = 4;
}
```

Selecting this check box converts the `if-elseif-else` pattern to the following `switch-case` statements:

```
switch (x) {
     case 1:
```

```
        y = 1; break;
      case 2:
        y = 2; break;
      case 3:
        y = 3; break;
      default:
        y = 4; break;
}
```

Off

Preserve `if-elseif-else` decision logic in generated code.

Command-Line Information

Parameter: `ConvertIfToSwitch`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Enhance Readability of Code for Flow Charts”
- “Enhance Code Readability for MATLAB Function Blocks”
- “Control Code Style”

Preserve extern keyword in function declarations

Description

Specify whether to include the `extern` keyword in function declarations in the generated code.

Note The `extern` keyword is optional for functions with external linkage. It is considered good programming practice to include the `extern` keyword in function declarations for code readability.

Category: Code Generation > Code Style

Settings

Default: on

 On

Include the `extern` keyword in function declarations in the generated code. For example, the generated code for the model `rtwdemo_hyperlinks` contains the following function declarations in `rtwdemo_hyperlinks.h`:

```
/* Model entry point functions */
extern void rtwdemo_hyperlinks_initialize(void);
extern void rtwdemo_hyperlinks_step(void);
```

The `extern` keyword explicitly indicates that the function has external linkage. The function definitions in this example are in the generated file `rtwdemo_hyperlinks.c`.

 Off

Remove the `extern` keyword from function declarations in the generated code.

Command-Line Information

Parameter: `PreserveExternInFcnDecls`
Type: character vector

Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2

Preserve static keyword in function declarations

Description

Specify whether to include the `static` keyword in function declarations in the generated code.

Category: Code Generation > Code Style

Settings

Default: on

 On

Include the `static` keyword in function declarations in the generated code. You can link different executables generated from different models that refer to locally scoped subsystem and utility functions with the same name. This parameter also impacts these functions:

- Stateflow graphical function
- Variant subsystem
- MATLAB subfunction
- Privately scoped Simulink function

When you select this parameter, the generated code is compliant with MISRA C:2012 Rule 8.10.

 Off

Remove the `static` keyword in function declarations in the generated code.

Dependency

- This parameter requires Embedded Coder license when you generate code.
- This parameter appears only for ERT-based targets.
- This parameter is enabled when you select `Compact/Compact(with separate data file)` file packaging.

Command-Line Information

Parameter: `PreserveStaticInFcnDecls`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution, ROM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- `MISRA C:2012 Rule 8.10`

Suppress generation of default cases for Stateflow switch statements if unreachable

Description

Specify whether to generate default cases for switch-case statements in the code for Stateflow charts. This optimization works on a per-model basis. It applies to the code generated for a state that has multiple substates. For a list of the state functions in the generated code, see “Inline State Functions in Generated Code” (Simulink Coder).

Category: Code Generation > Code Style

Settings

Default: on

 On

Do not generate the default case when it is unreachable. This setting enables better code coverage because every branch in the generated code is falsifiable.

 Off

Generate a default case whether or not it is reachable. This setting supports MISRA C compliance and provides a backup in case of RAM corruption.

For example, when the state has a nontrivial `entry` function, the following default case appears in the generated code for the `during` function:

```
default:
 entry_internal();
 break;
```

In this case, the code marks the corresponding substate as active.

Command-Line Information

Parameter: `SuppressUnreachableDefaultCases`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	On
Efficiency	On (execution, ROM), Noimpact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Suppress Generation of Default Cases for Unreachable Stateflow Switch Statements”

Replace multiplications by powers of two with signed bitwise shifts

Description

Specify whether to replace multiplications by powers of two with signed bitwise shifts. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. Clearing this option increases the likelihood of generating MISRA C compliant code.

Category: Code Generation > Code Style

Settings

Default: on

 On

Generate code that replaces multiplications by powers of two with signed bitwise shifts.

For example, when you select this option, multiplications by 8 are left-shifted in the generated code:

```
Y.Out1 = (U.In1 << ((int8_T)3));
```

Similarly, multiplications by 16 are left-shifted in the generated code:

```
Y.Out4 = (U.In2 << ((int8_T)4));
```

 Off

Do not allow replacement of multiplications by powers of two with signed shifts. Clearing this option supports MISRA C compliance.

For example, when you clear this option, multiplications by 8 are not replaced by bitwise shifts:

```
Y.Out1 = U.In1 * ((int64_T)8);
```

Similarly, multiplications by 16 are not replaced by bitwise shifts:

```
Y.Out4 = U.In2 * ((int32_T)16);
```

Command-Line Information

Parameter: `EnableSignedLeftShifts`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Replace Multiplication by Powers of Two with Signed Bitwise Shifts”

Allow right shifts on signed integers

Description

Specify whether to allow signed right bitwise shifts in the generated C/C++ code. Some coding standards, such as MISRA, do not allow bitwise operations on signed integers. Clearing this option increases the likelihood of generating MISRA-C:2004 compliant code.

Category: Code Generation > Code Style

Settings

Default: on

 On

Generate code that uses right bitwise shifts on signed integers.

For example, when you select this option, right shifts appear in the generated code.

```
i >>= 3
```

 Off

Do not allow right shifts on signed integers. Clearing this option supports MISRA C compliance.

For example, when you clear this option, right shifts are replaced with a function call.

```
i = asr_s32(i, 3U);
```

Command-Line Information

Parameter: `EnableSignedRightShifts`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Generate Code with Right Shifts on Signed Integers”

Casting modes

Description

Specify how the code generator casts data types for variables.

Category: Code Generation > Code Style

Settings

Default: `Nominal`

Nominal

Generate code that uses default C compiler data type casting.

```
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;
  rtDWork.X++;
  rtb_equal_to_count = (rtDWork.X != 16);
  if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
  {
    rtY.Output = rtU.Input << 1;
  }
```

Standards Compliant

Generate code that casts data types to conform to MISRA standards.

```
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;
     rtDWork.X++;
  rtb_equal_to_count = (boolean_T)(int32_T)((int32_T)rtDWork.X != (int32_T)16);
  if (((int32_T)rtb_equal_to_count) && (rtPrevZCSigState.Amplifier_Trig_ZCE !=
       POS_ZCSIG)) {
    rtY.Output = (int32_T)(uint32_T)((uint32_T)rtU.Input << (uint32_T)(int8_T)1);
  }
```

Explicit

Generate code that casts data type values explicitly.

```
/* Model step function */
void rtwdemo_rtwecintro_step(void)
{
  boolean_T rtb_equal_to_count;
  rtDWork.X = (uint8_T)(1U + (uint32_T)(int32_T)rtDWork.X);
  rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);
  if (((int32_T)rtb_equal_to_count) && ((int32_T)((int32_T)
        rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG))) {
    rtY.Output = rtU.Input << 1;
  }
```

Command-Line Information

Parameter: `CastingMode`
Type: character vector
Value: `'Nominal'` | `'Standards'` | `'Explicit'`
Default: `'Nominal'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Cast Expressions in Generated Code”
- “MISRA C Guidelines”

Indent style

Description

Specify style for the placement of braces in generated code.

Category: Code Generation > Code Style

Settings

Default: K&R

K&R

For blocks within a function, an opening brace is on the same line as its control statement. For example:

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;
  if (OverrunFlag) {
    rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
    return;
  }

  OverrunFlag = TRUE;
  rtwdemo_counter_step();
  OverrunFlag = FALSE;
}
```

Allman

For blocks within a function, an opening brace is on its own line at the same level of indentation as its control statement. For example:

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;
  if (OverrunFlag)
  {
    rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
    return;
  }
```

```
    OverrunFlag = TRUE;
    rtwdemo_counter_step();
    OverrunFlag = FALSE;
  }
```

Command-Line Information

Parameter: `IndentStyle`
Type: character vector
Value: `'K&R' |'Allman'`
Default: `'K&R'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Indentation Style in Generated Code”

Indent size

Description

Specify indent size for generated code.

Category: Code Generation > Code Style

Settings

Default: 2

Specify an integer value that indicates the number of characters per indent level. Possible values range from 2–8 characters.

Command-Line Information

Parameter: `IndentSize`
Type: integer
Value: integer from 2–8
Default: 2

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2

- “Control Indentation Style in Generated Code”

Newline style

Description

Specify the newline character in the generated code.

Category: Code Generation > Code Style

Settings

Default: `Default`

Default

Generates the newline character based on the operating system that the code is generated on.

LF (Line Feed)

Generates the Line Feed character as the newline character in the generated code. `"\n"` is inserted as the newline character.

CR+LF (Carriage Return + Line Feed)

Generates the Carriage Return + Line Feed character as the newline character in the generated code. `"\r\n"` is inserted as the newline character.

Command-Line Information

Parameter: `NewlineStyle`
Type: character vector
Value: `'Default'|'LF'|'CRLF'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Newline Style in Generated Code”

Maximum line width

Description

Specify the maximum line width for wrapping generated code.

Category: Code Generation > Code Style

Settings

Default: `80`

Specify an integer value that indicates the maximum number of columns in a single line of generated code. Possible values range from `50–1000` columns.

If the comments exceed the maximum line width specified, the tail comments are generated on a new line with right justification. Other types of comments are not wrapped:

- `#define` tail comments
- Simulink block comments
- Stateflow object comments
- Banner comments

Example

Here is generated code that is wrapped using the default **Maximum line width** value `80`:

```
/* Definition for custom storage class: Default */
real_T const_val[4] = { 1.0, 2.0, 3.0, 4.0 } ;
                  /* This parameter defines the vector of output index values */
```

The tail comments are generated on a new line with right justification.

Here is the same code wrapped with **Maximum line width** set to `120`:

```
/* Definition for custom storage class: Default */
real_T const_val[4] = { 1.0, 2.0, 3.0, 4.0 } ;/* This parameter defines the vector of output index values */
```

Command-Line Information

Parameter: `MaxLineWidth`
Type: integer
Value: integer from `50–1000`
Default: `80`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Code Style” on page 8-2
- “Control Code Style”

Code Generation Parameters: Data Type Replacement

Model Configuration Parameters: Code Generation Data Type Replacement

The **Code Generation > Data Type Replacement** category includes parameters for replacing built-in data type names with user-defined names in the generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Data Type Replacement** pane.

Parameter	Description
"Replace data type names in the generated code" on page 9-5	Specify whether to replace built-in data type names with user-defined data type names in generated code.
"Replacement Name: double" on page 9-7	Specify a name for `double` built-in data types in generated code.
"Replacement Name: single" on page 9-9	Specify a name for `single` built-in data types in generated code.
"Replacement Name: int32" on page 9-11	Specify a name for `int32_T` built-in data types in generated code.
"Replacement Name: int16" on page 9-13	Specify a name for `int16_T` built-in data types in generated code.
"Replacement Name: int8" on page 9-15	Specify a name for `int8_T` built-in data types in generated code.
"Replacement Name: uint32" on page 9-17	Specify a name for `uint32_T` built-in data types in generated code.
"Replacement Name: uint16" on page 9-19	Specify a name for `uint16_T` built-in data types in generated code.
"Replacement Name: uint8" on page 9-21	Specify a name for `uint8_T` built-in data types in generated code.
"Replacement Name: boolean" on page 9-23	Specify a name for `boolean_T` built-in data types in generated code.
"Replacement Name: int" on page 9-26	Specify a name for `int_T` built-in data types in generated code.
"Replacement Name: uint" on page 9-28	Specify a name for `uint_T` built-in data types in generated code.

Parameter	Description
“Replacement Name: char” on page 9-30	Specify a name for `char_T` built-in data types in generated code.
“Replacement Name: uint64” on page 9-32	Specify a name for `uint64_T` built-in data types in generated code.
“Replacement Name: int64” on page 9-34	Specify a name for `int64_T` built-in data types in generated code.

Configure Data Type Replacements Programmatically

To programmatically replace the built-in data type names for your model, adjust the `ReplacementTypes` model parameter, which is a structure. This example code shows how to modify the `ReplacementTypes` parameter to replace the built-in data type names `int8`, `uint8`, and `boolean` with the custom data type names `my_T_S8`, `my_T_U8`, and `my_T_BOOL`.

```
model = bdroot;
cs = getActiveConfigSet(model);
set_param(cs,'EnableUserReplacementTypes','on');

struc = get_param(cs,'ReplacementTypes');
struc.int8 = 'my_T_S8';
struc.uint8 = 'my_T_U8';
struc.boolean = 'my_T_BOOL';

set_param(cs,'ReplacementTypes',struc);
```

See Also

More About

- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- “Model Configuration”

Code Generation: Data Type Replacement Tab

Replace built-in data type names with user-defined replacement data type names in the generated code for your model.

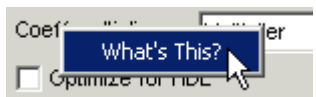
Configuration

This tab is visible only if you specify an ERT-based system target file (Simulink Coder).

1. Select **Replace data type names in the generated code**.
2. In the **Replacement Name** fields, selectively specify replacement data type names to use for built-in Simulink data types.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.



See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”

Replace data type names in the generated code

Description

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

Category: Code Generation > Data Type Replacement

Settings

Default: off

 On

Displays 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. Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

 Off

Uses Simulink Coder names for built-in Simulink data types in generated code.

Dependencies

This parameter enables replacement for all built-in data type name in the **Data type names** table with user-defined data type names in generated code.

Command-Line Information

Parameter: `EnableUserReplacementTypes`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: double

Description

Specify a name for `double` built-in data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `real_T`.

Specify a character vector for the code generator to use as a name for `double` built-in data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify `double` in the **Replacement Name** column.

To replace the **Code Generation Name** for `double` with an object:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `double`.
- For a `Simulink.NumericType` object, set the `DataTypeMode` object property to `Double`.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.double`
Type: character vector
Value: The **Simulink Name** , a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: single

Description

Specify a name for `single` built-in data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `real32_T`.

Specify a character vector for the code generator to use as a name for `single` built-in data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify `single` in the **Replacement Name** column.

To replace the **Code Generation Name** for `single` with an object:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `single`.
- For a `Simulink.NumericType` object, set the `DataTypeMode` object property to `Single`.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.single`
Type: character vector
Value: The **Simulink Name** , the name of a `Simulink.AliasType` object, or the name of a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: int32

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int32_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int32_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int32` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int32`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: int16

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int16_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int16_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int16`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: int8

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int8_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `int8_T`:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

- For a `Simulink.AliasType` object, set the `BaseType` object property to `int8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int8`

Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint32

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint32_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint32_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint32 c` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint32`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint16

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint16_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint16_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint16`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint8

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint8_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint8_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `uint8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint8`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: boolean

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `boolean_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

For ERT S-functions, the replacement data type can be only an 8-bit integer, `int8`, or `uint8`.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `boolean_T`:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `boolean`, `uint8` , `int8`, or `int`*n*, where *n* is the number of bits set for **Configuration Parameters** > **Hardware Implementation** > **Number of bits: int**. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- For a `Simulink.NumericType` object, to replace `real_T`, set the `DataTypeMode` object property to `Boolean`. Specify the name of the `Simulink.NumericType` object in the **Replacement Name** column.
- To use the Simulink Name built-in data type name which matches the Code Generation name, in the **Replacement Name** column, specify `uint8`, `int8`, or `int`*n*, where *n* is

the number of bits set for **Configuration Parameters > Hardware Implementation > Number of bits: int**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.boolean`
Type: character vector
Value: The **Simulink Name** , a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Replace boolean with Specific Integer Data Type”

- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`
- `Simulink.NumericType`

Replacement Name: int

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int_T`:

- For a `Simulink.AliasType` object

  Set the `BaseType` object property to `int`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- To use the **Simulink Name** for int_T, in the **Replacement Name** column, specify `int`*n*.

n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.AliasType`, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint

Description

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

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint_T`:

- For a `Simulink.AliasType` object

  Set the `BaseType` object property to `uint`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.
- To use the **Simulink Name** for `uint_T`, in the **Replacement Name** column, specify `uint`*n*.

n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint`
Type: character vector
Value: The **Simulink Name** or the name of a `Simulink.NumericType`, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: char

Description

Specify names for built-in Simulink data types in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `char_T`.

Specify character vectors for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `char_T`, create a `Simulink.AliasType` object in the Command Window.

Set the `BaseType` object property to `int`*n*. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column. *n* is the number of bits set for **Configuration Parameters** > **Hardware ImplementationNumber of bits: char**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.char`
Type: character vector

Value: The name of a `Simulink.AliasType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.AliasType`

Replacement Name: uint64

Description

Specify a name for a 64-bit unsigned integer Simulink data type in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `uint64_T`.

Specify character vectors for the code generator to use as names for 64-bit unsigned integer Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint64_T`:

- For a `Simulink.NumericType` object, set these properties:
  - `DataTypeMode` - `Fixed-point: binary point scaling`
  - `Signedness` - `Unsigned`
  - `WordLength` - `64`
  - `IsAlias` - `true`
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint64` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.

- The object is `Simulink.AliasType` .

Dependency

Replace data type names in the generated code on page 9-5 enables this parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint64`
Type: character vector
Value: The name of a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	A valid character vector
Efficiency	Noimpact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.NumericType`

Replacement Name: int64

Description

Specify a name for a 64-bit integer Simulink data type in generated code.

Category: Code Generation > Data Type Replacement

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `int64_T`.

Specify character vectors for the code generator to use as names for 64-bit integer Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int64_T`:

- For a `Simulink.NumericType` object, set these properties:
  - `DataTypeMode` - `Fixed-point: binary point scaling`
  - `Signedness` - `Signed`
  - `WordLength` - `64`
  - `IsAlias` - `true`
- To use the built-in data type name that matches the **Code Generation Name**, specify `int64` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.

- The object is `Simulink.AliasType`.

Dependency

Replace data type names in the generated code on page 9-5 enables **Replacement Name: int64** parameter.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int64`
Type: character vector
Value: The name of a `Simulink.NumericType` object, where the object exists in the base workspace.
Default: `''`

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	A valid character vector
Efficiency	Noimpact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Data Type Replacement” on page 9-2
- “Control Data Type Names in Generated Code”
- “Data Type Replacement Limitations”
- `Simulink.NumericType`

10

Memory Sections Parameters on the Code Generation Pane

Code Generation: Memory Sections Tab Overview

Insert comments and pragmas into the generated code for data and functions.

Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Package

Description

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

Category: Code Generation

Settings

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

Default: `---None---`

`---None---`

Suppresses memory sections.

`Simulink`

Applies the built-in Simulink package.

`mpt`

Applies the built-in mpt package.

Tip

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

Command-Line Information

Parameter: `MemSecPackage`
Type: character vector
Value: `'--- None ---'` | `'Simulink'` | `'mpt'`
Default: `'--- None ---'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Refresh package list

Description

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

Category: Code Generation

Tip

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

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Initialize/Terminate

Description

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

Category: Code Generation

Settings

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

Default: `Default`

`Default`

Suppresses the use of a memory section for Initialize, Start, and Terminate functions.

`memory-section-name`

Applies a memory section to Initialize, Start, and Terminate functions.

Command-Line Information

Parameter: `MemSecFuncInitTerm`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Execution

Description

Specify whether to apply a memory section to execution functions.

Category: Code Generation

Settings

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

Default: `Default`

`Default`

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

`memory-section-name`

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

Command-Line Information

Parameter: `MemSecFuncExecute`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Shared utility

Description

Specify whether to apply memory sections to shared utility functions.

Category: Code Generation

Settings

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

Default: `Default`

`Default`

Suppresses the use of memory sections for shared utility functions.

`memory-section-name`

Applies a memory section to shared utility functions, such as fixed-point functions, lookup table functions, and binary search functions.

Command-Line Information

Parameter: `MemSecFuncSharedUtil`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Constants

Description

Specify whether to apply a memory section to constants.

Category: Code Generation

Settings

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

Default: `Default`

`Default`

Suppresses the use of a memory section for constants.

`memory-section-name`

Applies a memory section to constants.

This parameter applies to the generated global data structures that contain:

- Constant parameters
- Constant block I/O

For basic information about the global data structures generated for models, see "Standard Data Structures in the Generated Code" (Simulink Coder).

Command-Line Information

Parameter: `MemSecDataConstants`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Inputs/Outputs

Description

Specify whether to apply a memory section to root input and output.

Category: Code Generation

Settings

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

Default: `Default`

`Default`

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

`memory-section-name`

Applies a memory section for root-level input and output.

This parameter applies to the generated global data structures that contain:

- Root-level inputs
- Root-level outputs

For basic information about the global data structures generated for models, see "Standard Data Structures in the Generated Code" (Simulink Coder).

Command-Line Information

Parameter: `MemSecDataIO`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Internal data

Description

Specify whether to apply a memory section to internal data.

Category: Code Generation

Settings

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

Default: `Default`

`Default`

Suppresses the use of a memory section for internal data.

`memory-section-name`

Applies a memory section for internal data.

This parameter applies to the generated global data structures that contain:

- Block I/O
- DWork vectors
- Zero-crossings

For basic information about the global data structures generated for models, see "Standard Data Structures in the Generated Code" (Simulink Coder).

Command-Line Information

Parameter: `MemSecDataInternal`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Parameters

Description

Specify whether to apply a memory section to parameters.

Category: Code Generation

Settings

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

Default: `Default`

`Default`

Suppress the use of a memory section for parameters.

`memory-section-name`

Apply memory section for parameters.

This parameter applies to the generated global data structure that contains block parameter data.

For basic information about the global data structures generated for models, see “Standard Data Structures in the Generated Code” (Simulink Coder).

Command-Line Information

Parameter: `MemSecDataParameters`
Type: character vector
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)
- Memory Sections

Validation results

Description

Display the results of memory section validation.

Category: Code Generation

Settings

The code generation software checks and reports whether the currently chosen package is on the MATLAB path and that the selected memory sections exist inside the package.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” (Simulink Coder)

11

Code Generation Parameters: Templates

Model Configuration Parameters: Code Generation Templates

The **Code Generation > Templates** category includes parameters for customizing the organization of your generated code. On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Templates** pane.

Parameter	Description
"Code templates: Source file (*.c) template" on page 11-5	Specify the code generation template (CGT) file to use when generating a source code file.
"Code templates: Header file (*.h) template" on page 11-7	Specify the code generation template (CGT) file to use when generating a code header file.
"Data templates: Source file (*.c) template" on page 11-9	Specify the code generation template (CGT) file to use when generating a data source file.
"Data templates: Header file (*.h) template" on page 11-11	Specify the code generation template (CGT) file to use when generating a data header file.
"File customization template" on page 11-13	Specify the custom file processing (CFP) template file to use when generating code.
"Generate an example main program" on page 11-15	Control whether to generate an example main program for a model.
"Target operating system" on page 11-18	Specify a target operating system to use when generating model-specific example main program module.

The following parameters on **Advanced parameters** section are infrequently used and have no other documentation.

Parameter	Description
`GenerateFullHeader`	Generate full header including time stamp. For GRT targets, this parameter is on the **Code Generation > Interface** pane.

Parameter	Description
`ERTCustomFileBanners`	If this option is cleared, the configurations for Code and Data templates are ignored.

See Also

More About

- “Model Configuration”

Code Generation: Templates Tab Overview

Customize the organization of your generated code.

Configuration

This tab appears only if you specify an ERT based system target file (Simulink Coder).

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2

Code templates: Source file (*.c) template

Description

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

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated source code files (`.c` or `.cpp`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTSrcFileBannerTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

Code templates: Header file (*.h) template

Description

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

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated header files (`.h`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTHdrFileBannerTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

Data templates: Source file (*.c) template

Description

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

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

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.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataSrcFileTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

Data templates: Header file (*.h) template

Description

Specify the code generation template (CGT) file to use when generating a data header file.

Category: Code Generation > Templates

Settings

Default: `ert_code_template.cgt`

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.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataHdrFileTemplate`
Type: character vector
Value: valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

File customization template

Description

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

Category: Code Generation > Templates

Settings

Default: `'example_file_process.tlc'`

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.

Command-Line Information

Parameter: `ERTCustomFileTemplate`
Type: character vector
Value: valid TLC file
Default: `'example_file_process.tlc'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- Selecting and Defining Templates
- Custom File Processing

Generate an example main program

Description

Control whether to generate an example main program for a model.

Category: Code Generation > Templates

Settings

Default: on

 On

Generates an 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 on whether your model is single-rate or multirate, and also on your model's solver mode (`SingleTasking` or `MultiTasking`).

 Off

Does not generate an example main program.

Note The software provides static versions of the main file, *`matlabroot`*`/rtw/c/src/common/rt_main.c` and *`matlabroot`*`/rtw/c/src/common/rt_cppclass_main.cpp`, as a basis for custom modifications. You can use either static main 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 code generator produces slightly different rate grouping code to maintain compatibility with an older static main module.

Dependencies

- This parameter enables **Target operating system**.
- You must enable this parameter and select `VxWorksExample` for **Target operating system** if you use VxWorks[3] library blocks.

Command-Line Information

Parameter: `GenerateSampleERTMain`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- “Deploy Generated Standalone Executable Programs To Target Hardware”

3. VxWorks is a registered trademark of Wind River Systems, Inc.

- Custom File Processing

Target operating system

Description

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

Category: Code Generation > Templates

Settings

Default: `BareBoardExample`

`BareBoardExample`

Generates a bareboard main program designed to run under control of a real-time clock, without a real-time operating system.

`VxWorksExample`

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

`NativeThreadsExample`

Generates a fully commented example showing how to deploy the threaded code under the host operating system. This option requires you to configure your model for concurrent execution.

Dependencies

- This parameter is enabled by **Generate an example main program**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `TargetOS`
Type: character vector
Value: `'BareBoardExample'` | `'VxWorksExample'` | `'NativeThreadsExample'`
Default: `'BareBoardExample'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Templates” on page 11-2
- “Deploy Generated Standalone Executable Programs To Target Hardware”
- Custom File Processing

Code Generation Parameters: Verification

Model Configuration Parameters: Code Generation Verification

The **Code Generation > Verification** category includes code verification and performance analysis parameters for SIL and PIL simulations. These parameters require an Embedded Coder license.

Parameter	Description
“Measure task execution time” on page 12-5	Measure execution times and generate metrics for tasks in generated code.
“Measure function execution times” on page 12-7	Measure execution times and generate metrics for functions inside generated code.
“Workspace variable” on page 12-9	Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.
“Save options” on page 12-11	Specify whether to save code profiling measurement and analysis data to base workspace.
“Third-party tool” on page 12-13	Specify a third-party tool for code coverage.
“Enable portable word sizes” on page 12-15	Allow portability across host and target processors that support different word sizes.
“Enable source-level debugging for SIL” on page 12-17	Allow debugging of generated code during a SIL simulation.

This parameter belongs to the **Advanced parameters** category.

Parameter	Description
“Create block” on page 5-2	Generate a SIL or PIL block.

See Also

More About

- “Numerical Equivalence Testing”
- “Code Execution Profiling”
- “Code Coverage”
- “Model Configuration”

Code Generation: Verification Tab Overview

Create SIL block and configure word size portability, code coverage for SIL testing, and code execution profiling

Configuration

This tab appears only if you specify an ERT–based system target file (Simulink Coder).

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “SIL and PIL Simulations”

Measure task execution time

Description

Measure execution times and generate metrics for tasks in generated code.

Category: Code Generation > Verification

Settings

Default: off

 On

During SIL and PIL simulations, collect execution-time measurements for tasks. The software obtains data from instrumentation in the SIL or PIL application.

 Off

Do not collect measurements of execution times

Dependencies

When you use this parameter, you must also specify a workspace variable. The software uses this variable to collect execution-time measurements.

In a model reference hierarchy, the top-model parameter value applies to the whole hierarchy. The software ignores the value of this parameter in referenced models.

Command-Line Information

Parameter: `CodeExecutionProfiling`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Measure function execution times

Description

Measure execution times and generate metrics for functions inside generated code.

Category: Code Generation > Verification

Settings

Default: `Off`

`Off`

No function-level instrumentation, so execution times for functions in generated code are not collected.

`Coarse (referenced models and subsystems only)`

Measure execution times only for function code generated from referenced models and subsystems.

`Detailed (all function call sites)`

Measure execution times for all functions in generated code.

Dependencies

To use this parameter, you must also select **Measure task execution time** for the top model of the model reference hierarchy.

For a model in a reference hierarchy, the software does not support simultaneous function execution-time measurement and code coverage.

Command-Line Information

Parameter: `CodeProfilingInstrumentation`
Type: character vector
Value: `'off'` | `'coarse'` | `'detailed'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Workspace variable

Description

Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.

Category: Code Generation > Verification

Settings

Default: `executionProfile`

When you run simulation, software generates specified workspace variable as an `coder.profile.ExecutionTime` object. To view and analyze execution profiles, use methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Command-Line Information

Parameter: `CodeExecutionProfileVariable`
Type: character vector
Value: valid MATLAB variable name
Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Save options

Description

Specify whether to save code profiling measurement and analysis data to base workspace.

Category: Code Generation > Verification

Settings

Default: `Summary data only`

`Summary data only`

Save only code profiling summary data to a `coder.profile.ExecutionTime` object in the base workspace. Use this option to limit the amount of data that the software saves to base workspace. For example, if you are concerned that your computer may not have enough memory to store the time measurements for a long simulation. The software calculates metrics for the code execution report as the simulation proceeds, without saving raw data to memory. To view these metrics, use the `coder.profile.ExecutionTime` `report` method.

Selecting this value disables the streaming of execution times to the Simulation Data Inspector during simulations.

`All data`

Save the code profiling measurement and analysis data to a `coder.profile.ExecutionTime` object in the base workspace. In addition to viewing the code execution report, this option allows you to analyze data using `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` methods.

Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Command-Line Information

Parameter: `CodeProfilingSaveOptions`

Type: character vector
Value: `'SummaryOnly'` | `'AllData'`
Default: `'SummaryOnly'`

Recommended Settings

Application	Setting
Debugging	`All data`
Traceability	`All data`
Efficiency	`Summary data only`
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Code Execution Profiling with SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Third-party tool

Description

Specify a third-party tool for code coverage.

Category: Code Generation > Verification

Settings

Default: `None (use Simulink Coverage)`

`None (use Simulink Coverage)`

No third-party tool specified for code coverage. You can use Simulink Coverage™ to analyze code coverage.

`BullseyeCoverage`

Specifies the BullseyeCoverage tool from Bullseye Testing Technology

`LDRA Testbed`

Specifies the LDRA Testbed® tool from LDRA Software Technology

Dependencies

Code coverage is not supported if the **Create block** configuration parameter is either `SIL` or `PIL`.

If you do not specify a third-party tool, **Configure Coverage** appears dimmed. Otherwise, click **Configure Coverage** to open the Code Coverage Settings dialog box.

Command-Line Information

Parameter: `CoverageTool` field of `CodeCoverageSettings`
Type: character vector
Value: `'None'` | `'BullseyeCoverage'` | `'LDRA Testbed'`
Default: `'None'`

Tip To access the `CoverageTool` value, type:

```
covSettings = get_param(gcs, 'CodeCoverageSettings');
covSettings.CoverageTool
```

Recommended Settings

Application	Setting
Debugging	`BullseyeCoverage` or `LDRA Testbed`
Traceability	`BullseyeCoverage` or `LDRA Testbed`
Efficiency	`None` (code coverage off)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Configure Code Coverage with Third-Party Tools”
- “Configure Code Coverage Programmatically”

Enable portable word sizes

Description

Allow portability across host and target processors that support different word sizes.

You can enable portable word sizes to support SIL testing of your generated code. For a SIL simulation, you use the top-model or Model block SIL simulation mode, or select `SIL` in the **Configuration Parameters** > **Create block** field.

Category: Code Generation > Verification

Settings

Default: off

 On

Generate conditional processing macros to 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. This option allows you to use the same generated code for software-in-the-loop (SIL) testing on the host platform and production deployment on the target platform. For example, you can perform SIL testing on a 32-bit host and deploy the code on a 16-bit target.

 Off

Does not generate portable code.

Dependencies

When you use this option, you should select **Test hardware is the same as production hardware** on the **Hardware Implementation** pane.

Command-Line Information

Parameter: `PortableWordSizes`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Configure Hardware Implementation Settings”

Enable source-level debugging for SIL

Description

Allow debugging of generated code during a SIL simulation.

Category: Code Generation > Verification

Settings

Default: off

 On

Source-level debugging is enabled.

 Off

Source-level debugging is disabled.

Command-Line Information

Parameter: `SILDebugging`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Verification” on page 12-2
- “Debug Generated Code During SIL Simulation”

Configuration Parameters

Recommended Settings Summary for Model Configuration Parameters

The following tables summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the ERT target. The Simulink Coder configuration parameters are documented in “Recommended Settings Summary for Model Configuration Parameters” (Simulink Coder). For additional details, click the links in the Configuration Parameter column.

Mapping of Application Requirements to the Optimization Pane

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Application lifespan (days) (Simulink)	No impact	No impact	Optimal finite value	`inf`	`auto`
Optimize using the specified minimum and maximum values (Simulink Coder)	Off	Off	On	No recommendation	Off
Remove root level I/O zero initialization (Simulink Coder)	No impact	No impact	On (GUI) `off` (command line) (execution, ROM), No impact (RAM)	Off	Off
Remove internal data zero initialization (Simulink Coder)	No impact	No impact	On (GUI) `off` (command line) (execution, ROM), No impact (RAM)	Off	Off
Remove code that protects against division arithmetic exceptions (Simulink Coder)	No impact	No impact	On (execution, ROM)	Off	Off

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Pack Boolean data into bitfields (Simulink Coder)	No impact	No Impact	Off (execution, ROM), On (RAM)	No impact	Off
Pass reusable subsystem outputs as (Simulink Coder)	No impact	No impact	`Structure reference` (ROM), `Individual arguments` (execution, RAM)	No impact	`Individual Arguments`

Mapping of Application Requirements to the Code Generation Pane: Memory Sections Parameters

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Package on page 10-3	No impact	No impact	No impact	No impact	`---None---`
Initialize/-Terminate on page 10-6	No impact	No impact	No impact	No impact	`Default`
Execution on page 10-8	No impact	No impact	No impact	No impact	`Default`
Shared utility on page 10-10	No impact	No impact	No impact	No impact	`Default`
Constants on page 10-12	No impact	No impact	No impact	No impact	`Default`
Inputs/Outputs on page 10-14	No impact	No impact	No impact	No impact	`Default`
Internal data on page 10-16	No impact	No impact	No impact	No impact	`Default`
Parameters on page 10-18	No impact	No impact	No impact	No impact	`Default`
Validation results on page 10-20	No impact	No impact	No impact	No impact	No package selected.

Mapping of Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Code-to-model (Simulink Coder)	On	On	No impact	No recommendation	On
Model-to-code (Simulink Coder)	On	On	No impact	No recommendation	On
Generate model Web view (Simulink Coder)	No impact	No impact	No impact	No impact	Off
Eliminated / virtual blocks (Simulink Coder)	On	On	No impact	No recommendation	On
Traceable Simulink blocks (Simulink Coder)	On	On	No impact	No recommendation	On
Traceable Stateflow objects (Simulink Coder)	On	On	No impact	No recommendation	On
Traceable MATLAB functions (Simulink Coder)	On	On	No impact	No recommendation	On
Static code metrics (Simulink Coder)	No impact	No impact	No impact	No impact	Off

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Summarize which blocks triggered code replacements (Simulink Coder)	No impact	No impact	No impact	No impact	Off

Mapping of Application Requirements to the Code Generation Pane: Comments Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Simulink block descriptions (Simulink Coder)	On	On	No impact	No impact	On
Simulink data object descriptions (Simulink Coder)	On	On	No impact	No impact	On
Custom comments (MPT objects only) (Simulink Coder)	On	On	No impact	No impact	Off
Custom comments function (Simulink Coder)	Valid file name	Valid file name	No impact	No impact	' '
Stateflow object descriptions (Simulink Coder)	On	On	No impact	No impact	On
Requirements in block comments (Simulink Coder)	On	On	No impact	No recommendation	Off

Mapping of Application Requirements to the Code Generation Pane: Symbols Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Global variables (Simulink Coder)	No impact	Use default	No impact	No recommendation	`$R$N$M`
Global types (Simulink Coder)	No impact	Use default	No impact	No recommendation	`&N$R$M_T`
Field name of global types (Simulink Coder)	No impact	Use default	No impact	No recommendation	`$N$M`
Subsystem methods (Simulink Coder)	No impact	Use default	No impact	No recommendation	`$R$N$M$F`
Subsystem method arguments (Simulink Coder)	No impact	Use default	No impact	No recommendation	`rt$I$N$M`
Local temporary variables (Simulink Coder)	No impact	Use default	No impact	No recommendation	`$N$M`
Local block output variables (Simulink Coder)	No impact	Use default	No impact	No recommendation	`rtb_$N$M`
Constant macros (Simulink Coder)	No impact	Use default	No impact	No recommendation	`$R$N$M`
Shared utilities identifier format (Simulink Coder)	No impact	Use default	No impact	No recommendation	`$N$C`

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Minimum mangle length (Simulink Coder)	No impact	1	No impact	No impact	1
Maximum identifier length (Simulink Coder)	Valid value	>30	No impact	>30	31
System-generated identifiers (Simulink Coder)	No impact	No impact	No impact	No impact	`Shortened`
Generate scalar inlined parameters as (Simulink Coder)	No impact	`Macros`	`Literals`	No impact	`Literals`
Use the same reserved names as Simulation Target (Simulink Coder)	No impact	No impact	No impact	No impact	Off
Shared checksum length (Simulink Coder)	No impact	No impact	No impact	No impact	8
EMX array utility functions identifier format (Simulink Coder)	No impact	No impact	No impact	No recommendation	`emx$M$N`

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
EMX array types identifier format (Simulink Coder)	No impact	No impact	No impact	No recommendation	`emxArray_$M$N`
Custom token text (Simulink Coder)	No impact	Set a custom string and use $U in symbols	No impact	No impact	`''`
#define naming (Simulink Coder)	No impact	`Force uppercase`	No impact	No impact	`None`
Parameter naming (Simulink Coder)	No impact	`Force uppercase`	No impact	No impact	`None`
Signal naming (Simulink Coder)	No impact	`Force uppercase`	No impact	No impact	`None`
MATLAB function (Simulink Coder)	No impact	No impact	No impact	No impact	`''`

Mapping of Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Support: floating-point numbers (Simulink Coder)	No impact	No impact	Off (GUI), `'on'` (command-line) for integer only	No impact	On (GUI), `'off'` (command-line)
Support complex numbers (Simulink Coder)	No impact	No impact	Off for real only	No impact	On
Support absolute time (Simulink Coder)	No impact	No impact	Off	No recommendation	On
Support continuous time (Simulink Coder)	No impact	No impact	Off (execution, ROM), No impact (RAM)	No recommendation	Off
Support non-inlined S-functions (Simulink Coder)	No impact	No impact	Off	No recommendation	Off
Support variable-size signals (Simulink Coder)	No impact	No impact	No impact	No recommendation	Off
Multiword type definitions (Simulink Coder)	No impact	No impact	No impact	No recommendation	`System defined`
Maximum word length (Simulink Coder)	No impact	No impact	No impact	No recommendation	256 for ERT targets 2048 for GRT targets

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Pass root-level I/O as (Simulink Coder)	No impact	No impact	No impact	No impact	`Individual arguments`
Use dynamic memory allocation for model initialization (Simulink Coder)	No impact	No impact	No impact	No recommendation	Off
Terminate function required (Simulink Coder)	No impact	No impact	No impact	No recommendation	On
Remove error status field in real-time model data structure (Simulink Coder)	Off	No impact	On	No recommendation	Off
Combine signal/ state structures (Simulink Coder)	Off	No impact	No impact	On	No impact
Parameter visibility (Simulink Coder)	No impact	No impact	No impact	No recommendation	`private`
Internal data visibility (Simulink Coder)	No impact	No impact	No impact	No recommendation	`private`
Parameter access (Simulink Coder)	`Inlined method`	`Inlined method`	`Inlined method`	No recommendation	`None`
Internal data access (Simulink Coder)	`Inlined method`	`Inlined method`	`Inlined method`	No recommendation	`None`

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
External I/O access (Simulink Coder)	`Inlined method`	`Inlined method`	`Inlined method`	No recommendation	`None`
Generate destructor (Simulink Coder)	No impact	No impact	No impact	No recommendation	On
Use dynamic memory allocation for model block instantiation (Simulink Coder)	No impact	No impact	On	No recommendation	Off

Mapping of Application Requirements to the Code Generation Pane: Verification Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Measure task execution time on page 12-5	On	On	Off	No recommendation	Off
Measure function execution times on page 12-7	On	On	Off	No recommendation	Off
Workspace variable on page 12-9	No impact	Valid MATLAB variable name	No impact	No impact	Off
Save options on page 12-11	`All data`	`All data`	`Summary data only`	No impact	`Summary data only`
Third-party tool on page 12-13	`BullseyeCoverage` or `LDRA Testbed`	`BullseyeCoverage` or `LDRA Testbed`	`None` (code coverage off)	No recommendation	`None` (code coverage off)
Enable portable word sizes on page 12-15	On	On	Off	No impact	Off
Enable source-level debugging for SIL on page 12-17	On	On	Off	No impact	Off

Mapping of Application Requirements to the Code Generation Pane: Code Style Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Parentheses level on page 8-5	`Nominal (Optimize for readability)`	`Nominal (Optimize for readability)`	`Minimum (Rely on C/C++ operators for precedence)`	No recommendation	`Nominal (Optimize for readability)`
Preserve operand order in expression on page 8-7	On	On	Off	No recommendation	Off
Preserve condition expression in if statement on page 8-9	On	On	Off	No recommendation	Off
Convert if-elseif-else patterns to switch-case statements on page 8-11	No impact	Off	On (execution, ROM), No impact (RAM)	No impact	On
Preserve extern keyword in function declarations on page 8-13	No impact	No impact	No impact	No impact	On
Preserve static keyword in function declarations on page 8-15	No impact	No impact	On (execution, ROM)	No impact	On
Suppress generation of default cases for Stateflow switch statements if unreachable on page 8-17	On	On	Off	No recommendation	On
Replace multiplications by powers of two with signed bitwise shifts on page 8-19	No impact	No impact	On	No impact	On

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Casting modes on page 8-23	Nominal	Nominal	Nominal	Standards Compliant	Nominal
Indent style on page 8-25	K&R	K&R	K&R	K&R	K&R
Indent size on page 8-27	2	2	2	2	2

Mapping of Application Requirements to the Code Generation Pane: Templates Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Code templates: Source file (*.c) template on page 11-5	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Code templates: Header file (*.h) template on page 11-7	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Data templates: Source file (*.c) template on page 11-9	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Data templates: Header file (*.h) template on page 11-11	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
File customization template on page 11-13	No impact	No impact	No impact	No impact	`example_file_process.tlc`
Generate an example main program on page 11-15	No impact	No impact	No impact	No impact	On
Target operating system on page 11-18	No impact	No impact	No impact	No impact	`BareBoard-Example`

Mapping of Application Requirements to the Code Generation Pane: Code Placement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Data definition on page 7-5	No impact	Valid value	No impact	No impact	Auto
Data definition filename on page 7-7	No impact	Valid value	No impact	No impact	`global.c`
Data declaration on page 7-9	No impact	Valid value	No impact	No impact	Auto
Data declaration filename on page 7-11	No impact	Valid value	No impact	No impact	`global.h`
#include file delimiter on page 7-13	No impact	Valid value	No impact	No impact	`off`
#include file delimiter on page 7-15	No impact	Valid value	No impact	No impact	Auto
Signal display level on page 7-17	No impact	Valid integer	No impact	No impact	10
Parameter tune level on page 7-19	No impact	Valid integer	No impact	No impact	10
File packaging format on page 7-21	No impact	No impact	No impact	No impact	`Modular`

Mapping of Application Requirements to the Code Generation Pane: Data Type Replacement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Replace data type names in the generated code on page 9-5	No impact	On	No impact	No impact	Off
Replacement Name on page 9-7	No impact	Valid character vector	No impact	No recommendation	' '

See Also

Related Examples

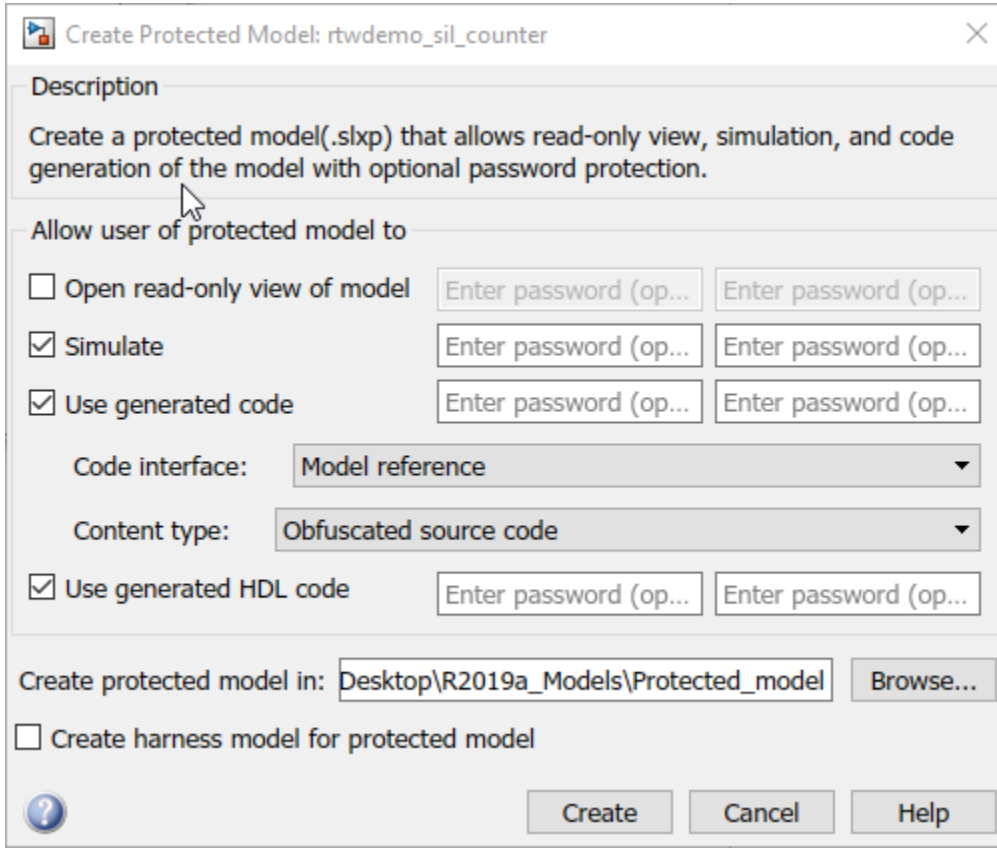
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor”
- “Code Generation Advisor Checks” (Simulink Coder)

14

Parameters for Creating Protected Models

Create Protected Model

This figure illustrates the various options in the Create Protected Model dialog box.



In this section...

In this section...

Create Protected Model: Overview

Create a protected model (.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.

To open the Create Protected Model dialog box, right-click the model block that references the model for which you want to generate protected model code. From the context menu, select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**.

See Also

- “Reference Protected Models from Third Parties” (Simulink)
- “Protect Models to Conceal Contents” (Simulink Coder)

Open read-only view of model

Share a view-only version of your protected model with optional password protection. View-only version includes the contents and block parameters of the model.

Settings

Default: Off

 On

Share a Web view of the protected model. For password protection, create and verify a password with a minimum of four characters.

 Off

Do not share a Web view of the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Simulate

Enable user to simulate a protected model with optional password-protection. Selecting **Simulate**:

- Enables protected model Simulation Report.
- Sets Mode to Accelerator. You can run normal, accelerator, and rapid accelerator mode simulations.
- Displays only binaries and headers.
- Enables code obfuscation.

Settings

Default: On

 On

User can simulate the protected model. For password protection, create and verify a password with a minimum of four characters.

 Off

User cannot simulate the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Use generated code

Allows user to generate code for the protected model with optional password protection. Selecting **Use generated code**:

- Enables Simulation Report and Code Generation Report for the protected model.
- Enables code generation.
- Enables support for simulation.

Settings

Default: Off

 On

User can generate code for the protected model. For password protection, create and verify a password with a minimum of four characters.

 Off

User cannot generate code for the protected model.

Dependencies

- To generate code, you must also select the **Simulate** check box.
- This parameter enables **Code interface** and **Content type**.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Requirements and Limitations”
- “Protect Models to Conceal Contents”

Code interface

Specify the interface for the generated code.

Settings

Default: `Model reference`

`Model reference`

Specifies the model reference interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations to verify code.

`Top model`

Specifies the standalone interface. Users of the protected model can run Model block SIL or PIL simulations to verify the protected model code.

Dependencies

- Requires an Embedded Coder license
- This parameter is enabled if you:
  - Specify an ERT (`ert.tlc`) system target file.
  - Select the **Use generated code** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Requirements and Limitations”
- “Protect Models to Conceal Contents”

Content type

Select the appearance of the generated code.

Settings

Default: `Obfuscated source code`

`Binaries`

Includes only compiled binaries for the generated code.

`Obfuscated source code`

Includes obfuscated source code.

`Readable source code`

Includes readable source code and readable code comments.

The options `Obfuscated source code` and `Readable source code` by default include only the minimal header files required to build the code with the chosen build settings. These options correspond to using the `Simulink.ModelReference.protect` with the `'OutputFormat'` option set to `'MinimalCode'`. To include header files found on the include path in the protected model, use the `Simulink.ModelReference.protect` function and set the `'OutputFormat'` option to `'AllReferencedHeaders'`.

The `Binaries` option corresponds to using the `Simulink.ModelReference.protect` function with the `'OutputFormat'` option set to `'CompiledBinaries'`.

Dependencies

This parameter is enabled by selecting the **Use generated code** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Use generated HDL code

Allows user to generate HDL code for the protected model with optional password protection. Selecting **Use generated HDL code**:

- Enables Simulation Report and HDL Code Generation Report for the protected model.
- Enables support for HDL code generation.
- Enables support for simulation.

Settings

Default: Off

 On

User can generate HDL code for the protected model. For password protection, create and verify a password with a minimum of eight characters.

 Off

User can simulate but cannot generate HDL code for the protected model.

Dependencies

To generate HDL code, you must also select the **Simulate** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Requirements and Limitations”
- “Protect Models to Conceal Contents”

Create protected model in

Specify the folder path for the protected model.

Settings

Default: Current working folder

Dependencies

A model that you protect must be available on the MATLAB path.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

Create harness model for protected model

Create a harness model for the protected model. The harness model contains only a Model block that references the protected model.

Settings

Default: Off

 On

Create a harness model for the protected model.

 Off

Do not create a harness model for the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models to Conceal Contents”

15

Model Advisor Checks

Embedded Coder Checks

In this section...

In this section...

Embedded Coder Checks Overview

Use Embedded Coder Model Advisor checks to configure your model for code generation.

See Also

- “Run Model Checks” (Simulink)
- “Simulink Checks” (Simulink)
- “Simulink Coder Checks” (Simulink Coder)

Check for blocks not recommended for C/C++ production code deployment

Check ID: `mathworks.codegen.PCGSupport`

Identify blocks not supported by code generation or not recommended for C/C++ production code deployment.

Description

This check partially identifies model constructs that are not recommended for C/C++ production code generation. For Simulink Coder and Embedded Coder, these model construct identities appear in tables of Simulink Block Support (Simulink Coder). If you are using blocks with support notes for code generation, review the information and follow the given advice.

Available with Simulink Check™ and Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Blocks and Products Supported for Code Generation” (Simulink Coder)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Identify lookup table blocks that generate expensive out-of-range checking code

Check ID: `mathworks.codegen.LUTRangeCheckCode`

Identify lookup table blocks that generate code to protect against out-of-range inputs for breakpoint or index values.

Description

This check verifies that the following blocks do not generate code to protect against inputs that fall outside the range of valid breakpoint values:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

This check also verifies that Interpolation Using Prelookup blocks do not generate code to protect against inputs that fall outside the range of valid index values.

Following the recommended actions increases both execution and ROM efficiency of the generated code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The lookup table block generates out-of-range checking code.	Change the setting on the block dialog box so that out-of-range checking code is not generated. • For the 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table, and Prelookup blocks, select the check box for **Remove protection against out-of-range input in generated code**. • For the Interpolation Using Prelookup block, select the check box for **Remove protection against out-of-range index in generated code**.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Action Results

Clicking **Modify** prevents lookup table blocks from generating out-of-range checking code, which makes the generated code more efficient.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- n-D Lookup Table
- Prelookup
- Interpolation Using Prelookup
- “Optimize Generated Code for Lookup Table Blocks” (Simulink)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check output types of logic blocks

Check ID: `mathworks.codegen.LogicBlockUseNonBooleanOutput`

Identify logic blocks that do not use `boolean` for the output data type.

Description

This check verifies that the output data type of the following blocks is `boolean`:

- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

Using output data type `boolean` increases execution efficiency of the generated code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The output data type of a logic block is not `boolean`.	In the block dialog box, set **Output data type** to `boolean`.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “What Is a Model Advisor Exclusion?” (Simulink Check)

Action Results

Clicking **Modify** forces logic blocks to use `boolean` as the output data type. If a logic block uses `uint8` for the output type, clicking **Modify** changes the output type to `boolean`.

Check the hardware implementation

Check ID: `mathworks.codegen.HWImplementation`

Identify inconsistent or underspecified hardware implementation settings

Description

The Simulink and Simulink Coder software require two sets of target specifications. The first set describes the final intended production target. The second set describes the currently selected target. If the configurations do not match, the code generator creates extra code to emulate the behavior of the production target. Inconsistencies or underspecification of hardware attributes can lead to inefficient or incorrect code generation for the target hardware.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Hardware implementation parameters are not set to recommended values.	In the Configuration Parameters dialog box, on the Hardware Implementation pane, specify the following parameters: • **Byte ordering** (`ProdEndianess`) • **Production hardware signed integer division rounds to** (`ProdIntDivRoundTo`) In the Configuration Parameters dialog box, specify the following parameters: • **Byte ordering in test hardware** (`TargetEndianess`) • **Test hardware signed integer division rounds to** (`TargetIntDivRoundTo`)
Hardware implementation **Production hardware** settings do not match **Test hardware** settings.	In the Configuration Parameters dialog box, consider selecting the **Test hardware is the same as production hardware** (`ProdEqTarget`) check box, or modify the settings to match.

See Also

"Run-Time Environment Configuration"

Identify questionable software environment specifications

Check ID: `mathworks.codegen.SWEnvironmentSpec`

Identify questionable software environment settings.

Description

- Support for some software environment settings can lead to inefficient code generation and nonoptimal results.

- Industry standards for C, such as ISO and MISRA, require identifiers to be unique within the first 31 characters.
- Stateflow charts with weak Simulink I/O data types lead to inefficient code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The maximum identifier length does not conform with industry standards for C.	In the Configuration Parameters dialog box, on the **Code Generation > Symbols** pane, set the **Maximum identifier length** (Simulink Coder) parameter to 31 characters.
In the Configuration Parameters dialog box, the parameters on the **Code Generation > Interface** pane are not set to recommended values.	In the Configuration Parameters dialog box, on the **Code Generation > Interface** (Simulink Coder) pane, clear the following parameters: • **Support: continuous time** • **Support: non-finite numbers** In the Configuration Parameters dialog box, clear **Support non-inlined S-functions**.
In the Configuration Parameters dialog box, the parameters on the **Code Generation > Symbols** pane are not set to recommended values.	In the Configuration Parameters dialog box, on the **Code Generation > Symbols** pane, set the **Generate scalar inlined parameters as** (Simulink Coder) parameter to `Literals`.
In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, **Support: variable-size signals** is selected. This might lead to inefficient code.	If you do not intend to support variable-sized signals, in the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, clear **Support: variable-size signals** (Simulink Coder).

Condition	Recommended Action
The model contains Stateflow charts with weak Simulink I/O data type specifications.	Select the Stateflow chart property **Use Strong Data Typing with Simulink I/O** (Stateflow). You might need to adjust the data types in your model after selecting the property.

Limitations

A Stateflow license is required when using Stateflow charts.

See Also

“Strong Data Typing with Simulink Inputs and Outputs” (Stateflow)

Identify questionable code instrumentation (data I/O)

Check ID: `mathworks.codegen.CodeInstrumentation`

Identify questionable code instrumentation.

Description

- Instrumentation of the generated code can cause nonoptimal results.
- Test points require global memory and are not optimal for production code generation.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Interface parameters are not set to recommended values.	In the Configuration Parameters dialog box, on the **Code Generation > Interface** (Simulink Coder) pane, set the parameters to the recommended values.
Blocks generate assertion code.	In the Configuration Parameters dialog box, set **Model Verification block enabling** (Simulink) to `Disable All` on a block-by-block basis or globally.

Condition	Recommended Action
Block output signals have one or more test points and, if you have an Embedded Coder license, the **Ignore test point signals** check box is cleared in the Configuration Parameters dialog box.	Remove test points from the specified block output signals. For each signal, in the Signal Properties dialog box (Simulink), clear the **Test point** check box. Alternatively, if the model is using an ERT-based system target file, select the **Ignore test point signals** check box in the Configuration Parameters dialog box to ignore test points during code generation.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “What Is a Model Advisor Exclusion?” (Simulink Check)

Identify blocks generating inefficient algorithms

Check ID: `mathworks.codegen.UseRowMajorAlgorithm`

Identify blocks generating inefficient algorithms.

Description

This check identifies the blocks that generate inefficient algorithms in the generated code based on the array layout of the model.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The configuration parameter **Array layout** (Simulink Coder) is set to `Column-major` for column-major code generation.	Disable the configuration parameter **Use algorithms optimized for row-major array layout** (Simulink).

Condition	Recommended Action
The configuration parameter **Array layout** is set to `Row-major` for row-major code generation.	Select the configuration parameter **Use algorithms optimized for row-major array layout**.

Capabilities and Limitations

- Analyzes content in masked subsystems.

See Also

- “Code Generation of Matrices and Arrays”

Check configuration parameters for MISRA C:2012

Check ID: `mathworks.misra.CodeGenSettings`

Identify configuration parameters that can impact MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Math and Data Types	
Configuration parameter **Use division for fixed-point net slope computation** is not set to `On` or `Use division for reciprocals of integers only`.	Set **Use division for fixed-point net slope computation** to `On` or `Use division for reciprocals of integers only`.
Inf or NaN block output is set to `None`	Set **Inf or NaN block output** to `warning` or `error`.

Condition	Recommended Action
Configuration parameter **Model Verification block enabling** is set to `Use local settings` or `Enable All`.	Set **Model Verification block enabling** to `Disable All`.
Configuration parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** to `error`.
Configuration parameter **Wrap on overflow** is set to `None`	Set configuration parameter **Wrap on overflow** to `warning` or `error`.
Hardware Implementation	
Configuration parameter **Production hardware signed integer division rounds to** is set to `Undefined`	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.
Configuration parameter **Shift right on a signed integer as arithmetic shift** is selected.	Clear **Shift right on a signed integer as arithmetic shift** .
Simulation Target	
Configuration parameter **Compile-time recursion limit for MATLAB functions** is set to a value other than `0` .	Set **Compile-time recursion limit for MATLAB functions** to `0` .
Configuration parameter **Dynamic memory allocation in MATLAB functions** is selected.	Clear **Dynamic memory allocation in MATLAB functions**.
Configuration parameter **Enable run-time recursion for MATLAB functions** is selected.	Clear **Enable run-time recursion for MATLAB functions**.
Code Generation	

Condition	Recommended Action
Configuration parameter **Bitfield declarator type specifier** is set to `uchar_T` when any of these parameters are selected: • **Pack Boolean data into bitfields** • **Use bitsets for storing state configuration** • **Use bitsets for storing Boolean data**	Set **Bitfield declarator type specifier** to `uint_T`.
Configuration parameter **Casting Modes** is not set to `Standards Compliant`.	Set **Casting Modes** to `Standards Compliant`.
Configuration parameter **Code replacement library**is not set to `None` or `AUTOSAR 4.0`.	Set **Code replacement library** to `None` or `AUTOSAR 4.0`
Configuration parameter **External mode** is selected.	Clear **External mode**.
Configuration parameter **Generate shared constants** is selected.	Clear **Generate shared constants**.
Configuration parameter **Include comments**is cleared.	Select **Include comments**.
Configuration parameter **MAT-file logging** is selected.	Clear **MAT-file logging**
For ERT-based target systems, configuration parameter **MATLAB user comments** is cleared.	Select **MATLAB user comments**.
A value for configuration parameter **Maximum identifier length** is not provided.	Set the value to the implementation-dependent limit. The default is `31`.
Configuration parameter **Parenthesis level** is not set to `Maximum (Specify precedence with parentheses)`.	Set **Parentheses level** to `Maximum (Specify precedence with parentheses)`.

Condition	Recommended Action
For ERT-based target systems, configuration parameter **Preserve static keyword in function declarations** is cleared when **File packaging format** is set to or `CompactCompactWithDataFile`	Select **Preserve static keyword in function declarations**.
Configuration parameter **Replace multiplications by powers of two with signed bitwise shifts** is selected.	Clear **Replace multiplications by powers of two with signed bitwise shifts**.
Configuration parameter **Shared code placement** is set to `Auto`.	Set **Shared code placement** to `Shared location`
For ERT-based target systems, configuration parameter **Support continuous time** is selected	Clear **Support continuous time**.
Configuration parameter **Support non-finite numbers** is selected.	Clear **Support non-finite numbers**
For ERT-based target systems, configuration parameter **Support non-inlined S-functions** is selected	Clear **Support non-inlined S-functions**.
Configuration parameter **System-generated identifiers** is set to `Classic`.	Set **System-generated identifiers** to `Shortened`.
Configuration parameter **System target file** is set to a GRT-based target.	Set **System target file** to an ERT-based target.
Configuration parameter **Use dynamic memory allocation for model initialization** is selected when **Code Interface Packaging** is set to `Reusable Function`.	Clear **Use dynamic memory allocation for model initialization**. Select only when **Code Interface Packaging** is set to `Reusable Function`.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

This check does not review referenced models.

See Also

- hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations” (Simulink)

Check for blocks not recommended for MISRA C:2012

Check ID: `mathworks.misra.BlkSupport`

Identify blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	Consider other interpolation and extrapolation methods for the Lookup Table blocks.

Condition	Recommended Action
Deprecated Lookup Table blocks were found in the model or subsystem. Specific blocks are: • Lookup Table • Lookup Table (2-D)	Consider replacing the deprecated Lookup Table blocks.
S-Function Builder blocks were found in the model or subsystem.	Consider replacing the S-Function Builder blocks with blocks recommended for production.
From Workspace blocks were found in the model or subsystem	Consider replacing the From Workspace blocks with blocks recommended for production.
String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing the String blocks with blocks recommended for production.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- hisl_0020: Blocks not recommended for MISRA C:2012 compliance
- na_0027: Use of only standard library blocks
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations” (Simulink)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for unsupported block names

Check ID: `mathworks.misra.BlockNames`

Identify block names containing `/`.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Block names containing `/` were found in the model or subsystem.	Remove `/` from the block name.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MISRA C:2012, Rule 3.1
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations” (Simulink)

Check usage of Assignment blocks

Check ID: `mathworks.misra.AssignmentBlocks`

Identify Assignment blocks that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.

Description

This check applies to the Assignment block that is available in the Simulink block library under **Simulink > Math Operations**.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem might contain Assignment blocks with incomplete array initialization that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.	Set block parameter **Action if any output element is not assigned** to one of the recommended values: • **Error**, if Assignment block is not in an Iterator subsystem. • **Warning**, if Assignment block is in an Iterator subsystem.

Capabilities and Limitations

- Runs on library models.

- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking. However, the following check condition is not supported because edit-time checking is unable to determine whether the Assignment block is an Iterator subsystem.

Set block parameter **Action if any output element is not assigned** to one of the recommended values:

- **Error**, if Assignment block is not in an Iterator subsystem.
- **Warning**, if Assignment block is in an Iterator subsystem.

See Also

- MISRA C:2012, Rule 9.1
- ISO/IEC TS 17961: 2013, uninitref
- CERT C, EXP33-C
- CWE, CWE-908
- “hisl_0029: Usage of Assignment blocks” (Simulink)
- “MISRA C Guidelines”
- “MISRA C:2012 Compliance Considerations” (Simulink)
- “Secure Coding Standards”

Check for switch case expressions without a default case

Check ID: `mathworks.misra.SwitchDefault`

Identify switch case expressions that do not have a default case.

Description

The check flags model objects that have switch case expressions without a default case.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

The check does not flag blocks without default cases if they are justified with a Polyspace® annotation. When you run the check, the **Blocks with justification** table lists blocks without default cases that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model object has a switch case expression without a default case.	For Switch Case blocks, consider selecting block parameter **Show default case** to explicitly specify a default case.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MISRA C:2012, Rule 16.4
- ISO/IEC TS 17961: 2013, swtchdflt
- CERT C, MSC01-C
- CWE, CWE-478
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Secure Coding Standards”

Check for missing error ports for AUTOSAR receiver interfaces

Check ID: `mathworks.misra.AutosarReceiverInterface`

Identify AUTOSAR receiver interface inports that do not have matching error ports.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags AUTOSAR receiver interfaces inports that are missing error ports. The following table identifies the AUTOSAR data access mode types for receiver interface ports that are flagged by the check when the corresponding error port is missing.

AUTOSAR Data Access Mode Type	Flagged by Check?
`ImplicitReceive`	Yes
`ExplicitReceive`	Yes
`QueuedExplicitReceive`	No
`ErrorStatus`	No
`ModeReceive`	No
`IsUpdated`	No
`EndToEndRead`	Yes
`ExplicitReceiveByVal`	No
`otherwise`	No

The check does not flag missing error ports when they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists the missing error ports that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
AUTOSAR receiver interface inport does not have a matching error port.	Add missing error port and map to the corresponding AUTOSAR receiver interface inport.

Condition	Recommended Action
AUTOSAR receiver interface ports do not have a matching error port when data access mode is `ImplicitReceive`, `ExplicitReceive`, or `EndToEndRead`.	Add missing error port and map to the corresponding AUTOSAR receiver interface inport.

Capabilities and Limitations

You can:

- Analyzes top layer/root level models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C: 2012, Directive 4.7
- “MISRA C Guidelines”
- “What Is a Model Advisor Exclusion?” (Simulink Check)
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Configure AUTOSAR Elements and Properties” (AUTOSAR Blockset)
- “AUTOSAR Component Configuration” (AUTOSAR Blockset)

Check bus object names that are used as bus element names

Check ID: `mathworks.misra.BusElementNames`

Identify bus object names that are used as bus element names.

Description

Using this check increases the likelihood of generating code for embedded applications that is compliant with MISRA C:2012. The check flags instances where a Simulink.Bus object name is used as the Simulink.Bus element name.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
A bus object name is being used as a bus element name.	Change either the flagged bus object name or the bus element name so that they are not identical.

See Also

- MISRA C:2012, Rule 5.6
- MISRA AC AGC, Rule 5.3
- “MISRA C Guidelines”

Check configuration parameters for secure coding standards

Check ID: `mathworks.security.CodeGenSettings`

Identify configuration parameters that might impact compliance with secure coding standards.

Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Diagnostics	
Configuration parameter **Inf or NaN block output** is set to `none`.	Set **Inf or NaN block output** to `warning` or `error`.
Configuration parameter **Model Verification block enabling** is set to `Use local settings` or `Enable All`.	Set **Model Verification block enabling** to `Disable All`.

Condition	Recommended Action
Configuration parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** to `error`.
Configuration parameter **Wrap on overflow** is set to `none`.	Set **Wrap on overflow** to `warning` or `error`.
Hardware Implementation	
Configuration parameter **Production hardware signed integer division rounds to** is set to `Undefined`.	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.
Configuration parameter **Shift right on a signed integer as arithmetic shift** is selected.	Clear **Shift right on a signed integer as arithmetic shift** .
Simulation Target	
Configuration parameter **Compile-time recursion limit for MATLAB functions** is set to a value other than `0` .	Set **Compile-time recursion limit for MATLAB functions** to `0` .
Configuration parameter **Dynamic memory allocation in MATLAB functions** is selected.	Clear **Dynamic memory allocation in MATLAB functions**.
Configuration parameter **Enable run-time recursion for MATLAB functions** is selected.	Clear **Enable run-time recursion for MATLAB functions**.
Code Generation	
Configuration parameter **Code replacement library** is not set to `None` or `AUTOSAR 4.0`.	Set **Code replacement library** to `None` or `AUTOSAR 4.0`.
Configuration parameter **External mode** is selected.	Clear **External mode**.
Configuration parameter **Include comments**is cleared.	Select **Include comments**.
Configuration parameter **MAT-file logging** is selected.	Clear **MAT-file logging**.

Condition	Recommended Action
For ERT-based target systems, configuration parameter **MATLAB user comments** is cleared.	Select **MATLAB user comments**.
Configuration parameter **Replace multiplications by powers of two with signed bitwise shifts** is selected.	Clear **Replace multiplications by powers of two with signed bitwise shifts**.
For ERT-based target systems, configuration parameter **Support continuous time** is selected	Clear **Support continuous time**.
Configuration parameter **Support non-finite numbers** is selected.	Clear **Support: non-finite numbers**
For ERT-based target systems, configuration parameter **Support non-inlined S-functions** is selected	Clear **Support non-inlined S-functions**.
Configuration parameter **System target file** is set to a GRT-based target.	Set **System target file** to an ERT-based target.
Configuration parameter **Use dynamic memory allocation for model initialization** is selected.	Clear **Use dynamic memory allocation for model initialization**.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

See Also

“Secure Coding Standards”

Check for blocks not recommended for secure coding standards

Check ID: `mathworks.security.BlockSupport`

Identify blocks not recommended for compliance with secure coding standards.

Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	Consider other interpolation and extrapolation methods for the Lookup Table blocks.
Deprecated Lookup Table blocks were found in the model or subsystem. Specific blocks are: • Lookup Table • Lookup Table (2-D)	Consider replacing the deprecated Lookup Table blocks.
S-Function Builder blocks were found in the model or subsystem.	Consider replacing the S-Function Builder blocks with blocks recommended for production.
From Workspace blocks were found in the model or subsystem	Consider replacing the From Workspace blocks with blocks recommended for production.

Condition	Recommended Action
String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing the String blocks with blocks recommended for production.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- na_0027: Use of only standard library blocks
- “What Is a Model Advisor Exclusion?” (Simulink Check)
- “Secure Coding Standards”

Identify questionable subsystem settings

Check ID: `mathworks.codegen.QuestionableSubsysSetting`

Identify questionable subsystem block settings.

Description

Subsystem blocks implemented as void-void functions in the generated code use global memory to store the subsystem I/O.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Subsystem blocks have the **Subsystem Parameters** > **Function packaging** option set to `Nonreusable function`.	Set the **Subsystem Parameters** > **Function packaging** parameter to `Auto`.
Subsystem blocks have the **Subsystem Parameters** > **Function packaging** option set to `Reusable function`.	Set the **Subsystem Parameters** > **Function packaging** parameter to `Auto`.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- Subsystem block
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for blocks not supported for row-major code generation

Check ID: `mathworks.codegen.RowMajorCodeGenSupport`

Check for blocks not supported for row-major code generation.

Description

This check identifies the blocks that are not supported for row-major code generation.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model interfaces with external data that is in row-major array layout.	Set the configuration parameter **Array layout** (Simulink Coder) to `Row-major`.

Capabilities and Limitations

- Analyzes content in masked subsystems.

See Also

- “Code Generation of Matrices and Arrays”

Identify TLC S-Functions with unset array layout

Check ID: `mathworks.codegen.RowMajorUnsetSFunction`

Identify TLC S-Functions with unset array layout.

Description

This check identifies S-functions that have `SSArrayLayout` set to `SS_UNSET`. By default, every S-function has `SSArrayLayout` property set to `SS_UNSET`. This setting disables the S-function for row-major code generation. When the configuration parameter **Array layout** (Simulink Coder) is set to `Row-major`, the Embedded Coder reports an error. You can turn off the error by changing the **External functions compatibility for row-major code generation** (Simulink Coder) to `warning` or `none`.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The configuration parameter **Array layout** is set to `Column-major` for column-major code generation.	Set the `SSArrayLayout` property to `Column-major`.
The configuration parameter **Array layout** is set to `Row-major` for row-major code generation.	Set the `SSArrayLayout` property to `Row-major`.

Capabilities and Limitations

- Analyzes content in all masked subsystems.

See Also

- “Code Generation of Matrices and Arrays”

Identify blocks that generate expensive fixed-point and saturation code

Check ID: `mathworks.codegen.BlockSpecificQuestionableFxptOperations`

Identify fixed-point operations that can lead to nonoptimal results.

Description

Certain block settings can lead to expensive fixed-point and saturation code.

Results and Recommended Actions

Conditions	Recommended Action
Blocks generate expensive saturation code.	Check whether your application requires setting **Function Block Parameters > Signal Attributes > Saturate on integer overflow**. Otherwise, clear the **Saturate on integer overflow** parameter for the most efficient implementation of the block in the generated code.

Conditions	Recommended Action
Product blocks are multiplying signals with mismatched slope adjustment factors. The net slope computation uses multiplication followed by shifts, which is inefficient for some target hardware.	Set the **Optimization > Use division for fixed-point net slope computation** parameter to `On`, or `Use division for reciprocals of integers only` if the net slope can be approximated by a fraction and division is more efficient than multiplication and shifts on the target hardware. **Note** This optimization takes place only if certain simplicity and accuracy conditions are met. For more information, see "Handle Net Slope Computation" (Fixed-Point Designer).
Product blocks are configured with a divide operation for the first input and a multiply operation for the second input.	Reverse the inputs so the multiply operation occurs first and the division operation occurs second.
Product blocks are configured to do multiple division operations.	Multiply all the denominator terms together, and then do a single division using cascading Product blocks.
Product blocks are configured to do many multiplication or division operations.	Split the operations across several blocks, with each block performing one multiplication or one division operation.
Protection code generated as part of the division operation is redundant.	Verify that your model cannot cause exceptions in division operations and then remove redundant protection code by setting the **Optimization > Remove code that protects against division arithmetic exceptions** (Simulink Coder) parameter in the Configuration Parameters dialog box.

Conditions	Recommended Action
The data type range of the inputs of Sum blocks exceeds the data type range of the output, which can cause overflow or saturation.	Change the output and accumulator data types so the range equals or exceeds all input ranges. For example, if the model has two inputs • `int8` (–128 to 127) • `uint8` (0 to 255) The data type range of the output and accumulator must equal or exceed –128 to 255. A `int16` (–32768 to 32767) data type meets this condition.
A Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output.	Change the data types so the inputs, outputs, and accumulator have the same slope adjustment factor.
The net sum of the Sum block input biases does not equal the bias of the output.	Change the bias of the output scaling, making the net bias adjustment zero.
The input and output of the MinMax block have different data types.	Change the data type of the input or output.
The input of the MinMax block has a different slope adjustment factor than the output.	Change the scaling of the input or the output.
The initial condition of the Discrete-Time Integrator block is used to initialize both the state and the output.	Set the **Function Block Parameters** > **Initial condition setting** parameter to `State (most efficient)`.
Parameter overflow occurred for the Compare to Zero block. This block uses the input data type to represent zero. The input data type cannot represent zero exactly, so the input value was compared to the closest representable value of zero.	Select an input data type that can represent zero.

Conditions	Recommended Action
Parameter overflow occurred for the following Compare to Constant block. This block uses the input data type to represent its **Constant value** parameter. The **Constant value** parameter is outside the range that the input data type can represent. The input signal was compared to the closest representable value of the **Constant value** parameter.	Choose an input data type that can represent the **Constant value** parameter or change the **Constant value** parameter to match the input data type.

Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.
- If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Identify Blocks that Generate Expensive Fixed-Point and Saturation Code” (Fixed-Point Designer)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for missing const qualifiers in model functions

Check ID: `mathworks.misra.ModelFunctionInterface`

Identify missing const qualifiers in input data pointers.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags input data pointers that do not have a const qualifier.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
A const qualifier is not defined for the input data pointer.	Consider adding a const qualifier to the input data pointer.

See Also

- MISRA C:2012, Rule 8.13
- “MISRA C Guidelines”

Identify questionable fixed-point operations

Check ID: `mathworks.codegen.QuestionableFxptOperations`

Identify fixed-point operations that can lead to nonoptimal results.

Description

Less efficient code can result from blocks that generate cumbersome multiplication and division operations, expensive conversion code, inefficiencies in lookup table blocks, and expensive comparison code.

Results and Recommended Actions

Conditions	Recommended Action
Integer division generated code is large.	In the Configuration Parameters dialog box, on the **Hardware Implementation** pane, set the **Production hardware signed integer division rounds to** parameter to the recommended value.
Lookup Table vector of input values is not evenly spaced.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.
Lookup Table vector of input values is not evenly spaced when quantized, but it is very close to being evenly spaced.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_evenspace_cleanup`.

Conditions	Recommended Action
Lookup Table vector of input values is evenly spaced, but the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.
For a Prelookup or n-D Lookup Table block, **Index search method** is `Evenly spaced points`. Breakpoint data does not have power of 2 spacing.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. Otherwise, in the block parameter dialog box, specify a different **Index search method** to avoid the computation-intensive division operation.
n-D Lookup Table breakpoint data is not evenly spaced and **Index search method** is not `Evenly spaced points`.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing and then set **Index search method** to `Evenly spaced points`.
n-D Lookup Table breakpoint data is evenly spaced and **Index search method** is `Evenly spaced points`. But the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.
n-D Lookup Table breakpoint data is evenly spaced, but the spacing is not a power of 2. Also, **Index search method** is not `Evenly spaced points`.	Set **Index search method** to `Evenly spaced points`. Also, if the data is nontunable, consider an even, power of 2 spacing.
n-D Lookup Table breakpoint data is evenly spaced, and the spacing is a power of 2. But the **Index search method** is not `Evenly spaced points`.	Set **Index search method** to `Evenly spaced points`.
Blocks require multiword operations in generated code.	Adjust the word lengths of inputs to operations so that they do not exceed the largest word size of your processor. For more information, see “Fixed-Point Multiword Operations In Generated Code” (Fixed-Point Designer).
Blocks require cumbersome multiplication.	Restrict multiplication operations: • So the product integer size is not larger than the target integer size. • To the recommended size.

Conditions	Recommended Action
Product blocks are multiplying signals with mismatched slope adjustment factors.	Change the scaling of the output so that its slope adjustment factor is the product of the input slope adjustment factors.
Blocks multiply signals with nonzero bias.	Insert a Data Type Conversion block before and after the block containing the multiplication operation.
The inputs of the Relational Operator block have different data types.	• Change the data type and scaling of the invariant input to match other inputs. • Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type.
The inputs of the Relational Operator block have different slope adjustment factors.	Change the scaling of either input.
The output of the Relational Operator block is constant. This might result in dead code which will be eliminated by Simulink Coder.	Review your model design and either remove the Relational Operator block or replace it with the constant.

Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.
- If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- 1-D Lookup Table
- n-D Lookup Table
- Prelookup
- “Identify Questionable Fixed-Point Operations” (Fixed-Point Designer)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Identify blocks that generate expensive rounding code

Check ID: `mathworks.codegen.ExpensiveSaturationRoundingCode`

Check for blocks that generate expensive rounding code.

Description

Generated rounding code is inefficient because of **Integer rounding mode** parameter setting.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Generated code is inefficient.	Set the **Function Block Parameters > Integer rounding mode** parameter to the recommended value.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “Identify Blocks that Generate Expensive Rounding Code” (Fixed-Point Designer)
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for bitwise operations on signed integers

Check ID: `mathworks.misra.CompliantCGIRConstructions`

Identify Simulink blocks that contain bitwise operations on signed integers.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

This check requires a Stateflow license when Stateflow is used in the model.

Results and Recommended Actions

Condition	Recommended Action
The model has blocks that contain bitwise operations on signed integers.	Consider using unsigned integers for bitwise operations.

Capabilities and Limitations

You can:

- The check assumes that code is generated for the whole model. When code is generated by a subsystem build or export functions, the check can product incorrect results.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “hisl_0060: Configuration parameters that improve MISRA C:2012 compliance” (Simulink)
- “MISRA C:2012 Compliance Considerations” (Simulink)
- “Secure Coding Standards”

Check for recursive function calls

Check ID: `mathworks.misra.RecursionCompliance`

Identify recursive function calls in Stateflow charts.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags charts that have recursive function calls.

Available with Embedded Coder and Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
Chart has a recursive function call.	Remove recursive function call.

See Also

- MISRA C:2012, Dir 17.2
- “Guidelines for Avoiding Unwanted Recursion in a Chart” (Stateflow)

Check for equality and inequality operations on floating-point values

Check ID: `mathworks.misra.CompareFloatEquality`

Identify equality and inequality operations on floating-point values.

Description

The check flags sources causing equality or inequality operations on floating-point values.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

The check does not flag blocks with equality or inequality operations on floating-point values if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists blocks with equality or inequality operations that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model object has an equality or inequality operation on a floating-point value.	Consider using non-floating-point values for equality or inequality operations.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Dir 1.1
- CERT C, FLP00-C
- CWE, CWE-697
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Secure Coding Standards”

Check integer word length

Check ID: `mathworks.misra.IntegerWordLengths`

Identify integer word lengths that do not comply with hardware implementation settings

Description

The check flags integers whose word lengths exceed the number of bits permitted via the hardware implementation settings.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model object contains integer word lengths that are not compliant with hardware implementation settings.	Update the integer so its length does not exceed the permitted number of bits. You can view the permitted number of bits in the Configuration Parameters dialog box, on the **Hardware Implementation** > **Device details** pane.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “MISRA C Guidelines”
- “What Is a Model Advisor Exclusion?” (Simulink Check)
- “Secure Coding Standards”

Check block names

Check ID: `mathworks.codegen.BlockNames`

Checks whether block names in the **Code Perspective** pane include invalid characters.

Description

This edit-time check evaluates the block names in the **Code Perspective** pane. The check reports invalid characters in block names, except for:

- Blocks that are ignored or not recommended for code generation
- Virtual Subsystem blocks

The check verifies that block names comply with these guidelines:

Form:

name:

- Does not start with a number
- Does not include spaces at the beginning of a block name
- Does not use double byte characters
- Carriage returns are allowed

Allowed Characters:

name:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9 _

Results and Recommended Actions

Condition	Recommended Action
The block name in the **Code Perspective** pane does not conform to the guidelines.	Update the block name to comply with the guidelines.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks.
- Analyzes content in masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- “Simulink Built-In Blocks That Support Code Generation”

16

Tools in Embedded Coder—Alphabetical List

Embedded Coder Dictionary

Create code definitions, which control code generation for model data and functions

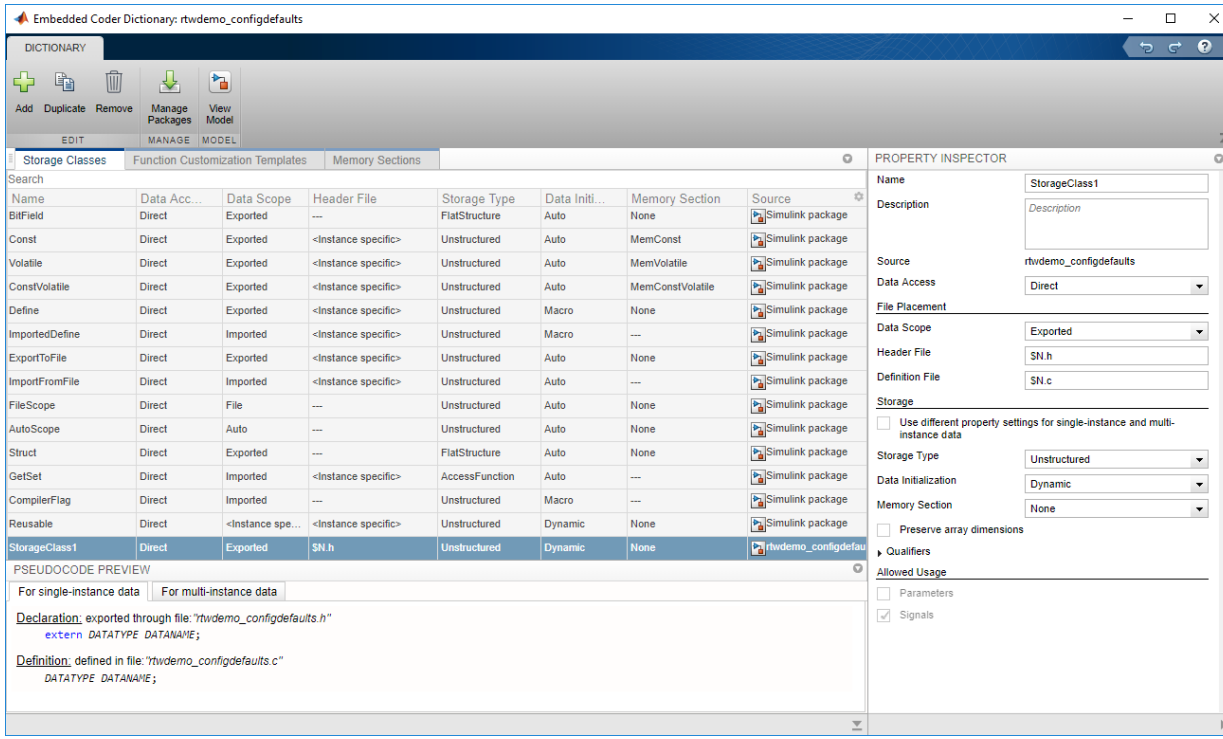
Description

The Embedded Coder Dictionary is a graphical interface for creating custom code definitions. By applying these definitions in models, you and your users can generate code that conforms to a specific software architecture by default. For example, you can create your own storage class, which you and your users can apply by default to a category of model data such as root-level inputs.

You can create these types of code definitions:

- Storage classes, which control the code generated for model data.
- Function customization templates, which control naming of model entry-point functions, such as `*model*_step`. The templates also apply memory sections to the entry-point functions.
- Memory sections, which control the placement of data and functions in memory. The generated code includes custom decorations, such as pragmas, whose syntax you specify.

For general information about creating code generation definitions, see “Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”.



The Embedded Coder Dictionary has a tab for each type of code definition. In each tab, you configure the properties of code definitions. Use the table to configure properties and compare definitions side by side. To access properties that do not appear in the table, use the Property Inspector. To verify results as you configure properties, use the pseudocode preview.

You can apply the definitions that you create in the dictionary to model elements by configuring model-wide default settings in the Code Mappings editor (see “Configure Default C Code Generation for Categories of Model Data and Functions”). To create storage classes and memory sections that you can use outside of the Code Mappings editor, use the Custom Storage Class Designer (see “Create Code Definitions to Override Default Settings”).

Open the Embedded Coder Dictionary

- To open an Embedded Coder Dictionary, use one of these techniques:

  - In the Code Mappings editor (see Code Mapping Editor), click the Embedded Coder Dictionary icon  .
  - In a model window, select **Code > C/C++ Code > Embedded Coder Dictionary**.

  The Embedded Coder Dictionary window displays code generation definitions that are stored in the model file. If the model is linked to a data dictionary, the window also displays definitions that are stored in that data dictionary or, if applicable, in a referenced dictionary. The **Source** column indicates where each definition is stored.
- To open the Embedded Coder Dictionary in a Simulink data dictionary, in the Model Explorer **Model Hierarchy** pane:

  1. Under the dictionary node, select the **Embedded Coder** node.

     If you do not see the node, right-click the dictionary node and select **Show Empty Sections**.
  2. In the Dialog pane (the right pane), click **Open Embedded Coder Dictionary**.

Examples

Create and Verify Custom Storage Class

In a model, create a storage class that aggregates internal model data, including block states, into a structure whose characteristics you can control. Then, verify the storage class by generating code from the model.

1. Open the example model `rtwdemo_roll`.

   ```
   rtwdemo_roll
   ```
2. Enable the Code perspective. In the Simulink Editor, select **Code > C/C++ Code > Configure Model in Code Perspective**.
3. Below the model diagram, in the Code Mappings editor, select the **Data Defaults** tab.

4. Click the Embedded Coder Dictionary icon  . The Embedded Coder Dictionary window displays code generation definitions that are stored in the model file.
5. In the Embedded Coder Dictionary window, click **Add**.
6. Select the new storage class that appears at the bottom of the list, `StorageClass1` . In the Property Inspector pane on the right, set the property values listed in this table.

Property	Value
Name	`InternalStruct`
Header File	`internalData_$R.h`
Definition File	`internalData_$R.c`
Storage Type	`Structured`
Structure Properties > **Type Name**	`internalData_T_$M`
Structure Properties > **Instance Name**	`internalData_$M`

After making your changes, in the bottom pane, verify that the pseudocode preview reflects what you expect.

7. Return to the Code Mappings editor. Select the **Internal data** row and set **Storage Class** to `InternalStruct`.
8. In the Configuration Parameters dialog box, on the **Code Generation** > **Code Placement** pane, set **File packaging format** to `Modular`.
9. Generate code.
10. In the Simulink Editor Code view, open and inspect the file `internalData_rtwdemo_roll.h`. The file defines the structure type `internalData_T_`, whose fields represent block states in the model.

```
/* Storage class 'InternalStruct', for system '<Root>' */
typedef struct {
  real32_T FixPtUnitDelay1_DSTATE;     /* '<S7>/FixPt Unit Delay1' */
  real32_T Integrator_DSTATE;          /* '<S1>/Integrator' */
  int8_T Integrator_PrevResetState;    /* '<S1>/Integrator' */
} internalData_T_;
```

The file also declares a global structure variable named `internalData_`.

```
/* Storage class 'InternalStruct' */
extern internalData_T_ internalData_;
```

11 Open and inspect the file `internalData_rtwdemo_roll.c`. The file allocates memory for `internalData_`.

```
/* Storage class 'InternalStruct' */
internalData_T_ internalData_;
```

Create Function Customization Template

With a function template, you can specify a rule that governs the names of generated entry-point functions. This technique helps save time and maintenance effort in a model with many entry-point functions, such as an export-function model or a multirate, multitasking model.

This example shows how to create a function template that specifies the naming rule `func_$N_$R`. `$N` is the base name of each generated function and `$R` is the name of the Simulink model.

1. Open the example model `rtwdemo_mrmtbb`.
2. Update the block diagram. This multitasking model has two execution rates, so the generated code includes two corresponding entry-point functions.
3. In the model, set model configuration parameter **System target file** to `ert.tlc`. To use a function customization template, you must use an ERT-based system target file.
4. In the Simulink Editor, enable Code perspective mode and open the Embedded Coder Dictionary.
5. In the Embedded Coder Dictionary, on the **Function Customization Templates** tab, click **Add**.
6. For the new function template, set these properties:

   - **Name** to `myFunctions`.
   - **Function Name** to `func_$N_$R`.

   After making your changes, verify that the pseudocode preview reflects what you expect.
7. On the **Function Defaults** tab, for the **Initialize/Terminate** and **Execution** rows, set **Function Customization Template** to `myFunctions`.
8. Generate code.
9. On the right side of the Code perspective, in the Code view pane, open and inspect the file `rtwdemo_mrmtbb.c`. The file defines the two execution functions, `func_step0_rtwdemo_mrmtbb` and `func_step1_rtwdemo_mrmtbb`, whose names conform to the rule that you specified in the function template.

Create Memory Section

For an example that shows how to create a memory section, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

Create Storage Class for Use with Statically and Dynamically Initialized Data

This example shows how to create a storage class that places global variable definitions and declarations in files whose names depend on the model name. You create two copies of the storage class so that you can use one copy with parameter data (the data category **Local parameters**) and one copy with other data.

Typically, the generated code initializes parameter data statically, outside any function, and initializes other data dynamically, in the model initialization function. When you create a storage class by using the Custom Storage Class Designer or an Embedded Coder Dictionary, you set the **Data Initialization** property to specify the initialization mechanism.

In an Embedded Coder Dictionary, for each storage class, you must select `Static` or `Dynamic`. Consider creating one copy of the storage class for parameter data (`Static`) and one copy for other data (`Dynamic`).

Create Storage Class

1. Open example model `rtwdemo_roll`.
2. Enable the Simulink Editor Code perspective mode.
3. Below the model diagram, in the Code Mappings editor, click the Embedded Coder Dictionary icon.
4. In the Embedded Coder Dictionary, click **Add**.
5. For the new storage class, set these properties:

   - **Name** to `SigsStates`
   - **Header File** to `$R_my_data.h`
   - **Definition File** to `$R_my_data.c`

   By default, the **Data Initialization** property is set to `Dynamic`, which means the storage class is suitable for use with signals, states, and data stores.

After making your changes, verify that the pseudocode preview reflects what you expect.

6 Click **Duplicate**. A new storage class, `SigsStates_copy`, appears.
7 For the new storage class, set these properties:

- **Name** to `Params`
- **Data Initialization** to `Static`

After making your changes, verify that the pseudocode preview reflects what you expect.

Apply Storage Class and Generate Code

1 Return to the Code Mappings editor.
2 On the **Data Defaults** tab, for the **Local parameters** row, in the **Storage Class** column, select `Params`.
3 For the **Internal data** row, set **Storage Class** to `SigsStates`.
4 Configure some parameter data elements in the model so that optimizations do not eliminate them from the generated code. Open the Model Data Editor.
5 Select the **Parameters** tab.
6 In the model, navigate into the `BasicRollMode` subsystem.
7 Update the block diagram. Now, the data table contains rows that correspond to workspace variables used by the model.
8 Next to the **Filter contents** box, activate the **Filter using selection** button.
9 In the model, select the three Gain blocks.
10 Update the diagram.
11 In the Model Data Editor, in the data table, select the three rows that correspond to variables `dispGain`, `intGain`, and `rateGain` in the model workspace.
12 For each variable, in the **Storage Class** column, select `Convert to parameter object`.

The Model Data Editor converts the workspace variables to `Simulink.Parameter` objects. The new objects use the storage class `Model default`, which means they acquire the default storage class that you specified for **Local parameters** in the Code Mappings editor.
13 In the Configuration Parameters dialog box, on the **Code Generation** > **Code Placement** pane, set **File packaging format** to `Modular`.
14 Generate code.
15 In the Code view, open and inspect the files `rtwdemo_roll_my_data.c` and `rtwdemo_roll_my_data.h`. These files define and declare global variables that

correspond to the parameter objects and some block states, such as the state of the Integrator block in the `BasicRollMode` subsystem.

```
/* Storage class 'SigsStates' */
real32_T rtFixPtUnitDelay1_DSTATE;
real32_T rtIntegrator_DSTATE;
int8_T rtIntegrator_PrevResetState;

/* Storage class 'Params' */
real32_T dispGain = 0.75F;
real32_T intGain = 0.5F;
real32_T rateGain = 2.0F;
```

Refer to Code Generation Definitions in a Package

You can configure an Embedded Coder Dictionary to refer to code generation definitions that you store in a package (see “Create Code Definitions to Override Default Settings”). Those definitions then appear available for selection in the Code Mappings editor. In this example, you configure the Embedded Coder Dictionary in `rtwdemo_roll` to refer to definitions stored in the built-in example package `ECoderDemos`.

1. Open the Embedded Coder Dictionary for `rtwdemo_roll`. For instructions, see “Create and Verify Custom Storage Class” on page 16-4.
2. In the Embedded Coder Dictionary window, click **Manage Packages**.
3. In the Manage Packages dialog box, click **Refresh**. Wait until more options appear in the **Select package** drop-down list.
4. Set **Select package** to `ECoderDemos` and click **Load**.

   In the Embedded Coder Dictionary window, on the **Storage Classes** tab, the table shows the storage classes defined in the `ECoderDemos` package. Now, in `rtwdemo_roll`, you can select these storage classes in the Code Mappings editor on the **Data Defaults** tab.
5. To unload the package, in the Manage Packages dialog box, select the package in the **Select package** drop-down list and click **Unload**.

Share Code Generation Definitions Between Models by Using Simulink Data Dictionary

For an example that shows how to share code generation definitions between models by using data dictionaries, see “Share Embedded Coder Dictionary Definition Between Models”.

Configure Default Code Mappings in a Shared Coder Dictionary

For an example that shows how to configure default code mappings in a shared Embedded Coder Dictionary, see “Configure Default Code Mapping in a Shared Dictionary”.

Parameters

These properties appear in the Property Inspector pane of the Embedded Coder Dictionary window. In the table, some properties appear as columns to facilitate batch editing.

Storage Classes

`Name` — Name of storage class
`StorageClass1` (default) | text

Name of the storage class. The name must be unique among the storage classes in the dictionary.

For lists of built-in and example storage classes that Simulink provides, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

`Description` — Purpose and functionality of storage class
text

Custom text that you can use to describe the purpose and functionality of the storage class.

`Source` — Location of storage class definition
text

This property is read-only.

The location of the storage class definition.

- `Built-in` — Provided by Simulink.
- Model name — Defined in a Simulink model.

- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?” (Simulink)).
- Package name — Defined in the Simulinkpackage or in a custom package (see “Create Custom Storage Classes by Using the Custom Storage Class Designer”).

`Data Access` — Specification to access the data

`Direct` (default) | `Function`

Specification to access data associated with the model. Access the data directly (`Direct`) or through customizable `get` and `set` functions (`Function`). For more information, see “Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary”.

Dependencies

- Setting this property to `Function`:
  - Sets **Data Access** to `Imported`.
  - Means that you cannot specify multi-instance properties.
  - Enables these properties:
    - **Access Mode**
    - **Allowed Access**
    - **Name of Getter**
    - **Name of Setter**
  - Disables the **Preserve array dimensions** property. To preserve dimensions of multidimensional arrays in the generated code, set **Data Access** to `Direct`.

`Data Scope` — Specification to generate data definition

`Exported` (default) | `Imported`

Specification that the generated code define the data (`Exported`) or import (`Imported`) the data definition from external code. Built-in storage classes and storage classes in packages such as Simulink can use other scope options, such as `File`.

Dependencies

- Setting this property to `Imported`:

  - Disables **Definition File**. To include your external source code file in the build process, use model configuration parameters. For an example, see “Configure Data Interface”.
  - Means that you cannot set **Header File** to `$N.h`, though you can use the `$N` token.
- To set this property to `Exported`, you must use one of the tokens `$N` or `$R` in the value of **Header File**.

`Header File` — Name of header file that declares data

`$N.h` (default) | text

Name of the header file that declares the data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Name of associated data element
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in “Identifier Format Control”

Dependencies

- If you set **Data Scope** to `Exported`, you must use one of the tokens `$R` or `$N` in the value of this property.
- If you set **Data Scope** to `Imported`, you cannot set the value of this property to `$N.h`, but you can use the `$N` token.

`Definition File` — Name of source file that defines data

`$N.c` (default) | text

Name of the source file that defines the data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model

Token	Description
`$N`	Name of associated data element
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in "Identifier Format Control"

Dependencies

Setting **Data Scope** to `Imported` disables **Definition File**. To include your external source code file in the build process, use model configuration parameters. For an example, see "Configure Data Interface".

`Access Mode` — Specification to access data through functions

`Value` (default) | `Pointer`

Specification for the storage class to access data associated with the model through functions by using `Value` or `Pointer`. For more information, see "Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary".

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Allowed Access` — Specification to allow access to data through functions

`Read/Write` (default) | `Read Only` | `Write Only`

Specification for the storage class to allow read and write (`Read/Write`), read-only (`Read Only`), or write-only (`Write Only`) access to the data.

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Name of Getter` — Name of the get function that fetches the associated data

`get_$N$M` (default) | text

Name of the `get` function that fetches the associated data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$N`	Name of associated data element (required)
`$R`	Name of root model
`$M`	Mangle text that ensures uniqueness
`$U`	User token text. See “Identifier Format Control”.

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Name of Setter` — Name of the `set` function that modifies the associated data
`set_$N$M` (default) | text

Name of the `set` function that fetches the modifies data, specified as a name or naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$N`	Name of associated data element (required)
`$R`	Name of root model
`$M`	Mangle text that ensures uniqueness
`$U`	User token text. See “Identifier Format Control”.

Dependencies

This property is enabled only when you set **Data Access** to `Function`.

`Use different property settings for single-instance and multi-instance data` — Specification to assign separate storage settings
`off` (default) | `on`

Specification for the storage class to use either the storage settings that you specify in the **Single-instance storage** section or the storage settings that you specify in the **Multi-instance storage** section. When you apply the storage class to a data item, the Embedded Coder Dictionary determines if it is a single-instance storage class or a multi-instance storage class by the type of data and by the context of the model within the model reference hierarchy.

Dependencies

Selecting this property enables the sections **Single-instance storage** and **Multi-instance storage**. The properties **Storage Type**, **Type Name**, and **Instance Name** appear in both the **Single-instance storage** and **Multi-instance storage** sections.

`Storage Type` — Specification to aggregate data into a structure

`Unstructured` (default) | `Structured`

Specification to aggregate the data that uses the storage class into a structure in the generated code. Each data element appears in the code as a field of the structure. To create a structure, use `Structured`.

Dependencies

Setting this property to `Structured` enables **Type Name** and **Instance Name**.

`Type Name` — Name of structure type

`$R$N$G$M` (default) | text

Name of the structure type in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in “Identifier Format Control”
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting **Storage Type** to `Structured` enables this property.

`Instance Name` — Name of structure variable

`$N$G$M` (default) | text

Name of the structure variable in the generated code, specified as a name or a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$G`	Name of storage class
`$U`	User token text, which you specify for a model as described in "Identifier Format Control"
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

Setting **Storage Type** to `Structured` enables this property.

`Data Initialization` — How to initialize data

`Dynamic` (default) | `Static` | `None`

Specification that the generated code initialize the data.

- `Dynamic` — The generated code initializes the data as part of the model initialization entry-point function.
- `Static` — The generated code initializes the data in the same statement that defines and allocates memory for the data. The assignment statement appears at the top of a `.c` or `.cpp` source file, outside of a function.
- `None` — The generated code does not initialize the data.

Dependencies

- If you select **Const**, you cannot set this property to `Dynamic`.
- Setting this property to `Dynamic` disables **Const**.

`Memory Section` — Location in memory to allocate data

`None` (default) | existing memory section

Location in memory to allocate data, specified as a memory section that exists in the Embedded Coder Dictionary on the **Memory Sections** tab. For information about

memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

`Preserve array dimensions` — Specification to preserve dimensions of multidimensional arrays
`off` (default) | `on`

Specification for the storage class to preserve dimensions of multidimensional arrays in the generated code. For more information, see “Preserve Dimensions of Multidimensional Arrays in Generated Code”.

`Const` — Specification to apply `const` qualifier
`off` (default) | `on`

Specification to apply the `const` qualifier to the data.

Dependencies

- If you select this property, you cannot set **Data Initialization** to `Dynamic`.
- Setting **Data Initialization** to `Dynamic` disables this property.

`Volatile` — Specification to apply `volatile` qualifier
`off` (default) | `on`

Specification to apply the `volatile` qualifier to the data.

`Other Qualifier` — Specification to apply a custom qualifier
text

Specification to apply a custom qualifier to the data. For example, some memory architectures support qualifiers `far` and `huge`.

Do not use this property to apply the keyword `static`. Instead, use the built-in storage class `FileScope`, which you cannot apply with the Code Mappings editor. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

`Parameters` — Whether to allow usage with model parameters
`off` (default) | `on`

Specification indicating whether to allow usage of the storage class with model parameters.

Dependencies

- Setting **Data Initialization** to `Static` enables this property.
- Setting **Data Initialization** to `Dynamic` disables this property.
- To set the value of this property, set **Data Initialization** to `None`.

`Signals` — Whether to allow usage with model signals

`on` (default) | `off`

Specification indicating whether to allow usage of the storage class with model signals.

Dependencies

- Setting **Data Initialization** to `Dynamic` enables this property.
- Setting **Data Initialization** to `Static` disables this property.
- To set the value of this property, set **Data Initialization** to `None`.

Function Customization Templates

`Name` — Name of function template

`FunctionTemplate1` (default) | text

Name of the template. The name must be unique among the function templates in the dictionary. Embedded Coder provides the built-in templates listed in this table.

Template	Description
`ModelFunction`	In the Code Mappings editor, use for entry-point functions for initialization, execution, termination, and reset (see “Configure Default Code Generation for Functions”)
`UtilityFunction`	In the Code Mappings editor, use for shared utility functions (see “Configure Default Code Generation for Functions”)

`Description` — Purpose and functionality of function template

text

Custom text that you can use to describe the purpose and functionality of the function template.

`Source` — Location of function template definition

text

This property is read-only.

The location of the function template definition.

- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?” (Simulink)).

`Function Name` — Names of generated functions

`$R$N` (default) | text

Names of the functions in the generated code, specified as a naming rule. A naming rule includes a combination of text and tokens. Valid tokens are listed in this table.

Token	Description
`$R`	Name of root model
`$N`	Base name of associated function, such as `step`
`$U`	User token text, which you specify for a model as described in “Identifier Format Control”
`$C`	For shared utility functions, a checksum inserted to avoid name collisions
`$M`	Name-mangling text inserted, if necessary, to avoid name collisions

`Memory Section` — Location in memory to allocate function

`None` (default) | existing memory section

Location in memory to allocate function, specified as a memory section that exists in the Embedded Coder Dictionary on the **Memory Sections** tab. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

Memory Sections

`Name` — Name of memory section

text

Name of the memory section. The name must be unique among the memory sections in the dictionary. Embedded Coder provides the built-in memory sections listed in this table.

Memory Section	Description
`MemConst`	Apply the storage type qualifier `const` to the data.
`MemVolatile`	Apply the storage type qualifier `volatile` to the data.
`MemConstVolatile`	Apply the storage type qualifiers `const` and `volatile` to the data.

`Description` — Purpose and functionality of memory section
text

Custom text that you can use to describe the purpose and functionality of the memory section.

`Source` — Location of memory section definition
text

This property is read-only.

The location of the memory section definition.

- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary (see “What Is a Data Dictionary?” (Simulink)).
- Package name — Defined in the Simulinkpackage or in a custom package (see “Create Code Definitions to Override Default Settings”).

`Comment` — Comment to insert in the generated code
text

Code comment that the code generator includes with the pragmas or other decorations that you specify with **Pre Statement** and **Post Statement**.

`Pre Statement` — Code to insert before data or function code
text

Code, such as pragmas, to insert before the definitions and declarations of the data or functions that are in the memory section.

When you set **Statements Surround** to `Each variable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

`Post Statement` — Code to insert after data or function code
text

Code, such as pragmas, to insert after the definitions and declarations of the data or functions that are in the memory section.

When you set **Statements Surround** to `Each variable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

`Statements Surround` — Specification to wrap data and functions separately or in a group
`Each variable` (default) | `Group of variables`

Specification to insert code statements (**Pre Statement** and **Post Statement**):

- Around each variable and function that uses the memory section. Select `Each variable`.
- Once, around the entire memory section. The generated code aggregates the variable and function definitions into a contiguous code block and surrounds the block with the statements. Select `Group of variables`.

Limitations

- You cannot create or modify code generation definitions programmatically. However, you can delete, copy, and move code definitions between models and data dictionaries by using these functions:
  - `coder.dictionary.copy`
  - `coder.dictionary.move`
  - `coder.dictionary.remove`
- A storage class or function customization template that you create in an Embedded Coder Dictionary cannot use a memory section that you load from a package (as described in “Refer to Code Generation Definitions in a Package” on page 16-9). Use a memory section defined in the Embedded Coder Dictionary.
- You cannot create code generation definitions in a `.mdl` model file.
- For additional limitations for code generation definitions in the Embedded Coder Dictionary of a data dictionary (`.sldd` file), see “Deploy Code Generation Definitions”.

See Also

Code Mapping Editor

Topics

“Define Storage Classes, Memory Sections, and Function Templates for Software Architecture”
“Deploy Code Generation Definitions”
“Conform to Software Architecture by Sharing and Copying Default Code Generation Settings Between Models”
“Flexible Storage Class for Different Model Hierarchy Contexts”

Introduced in R2018a

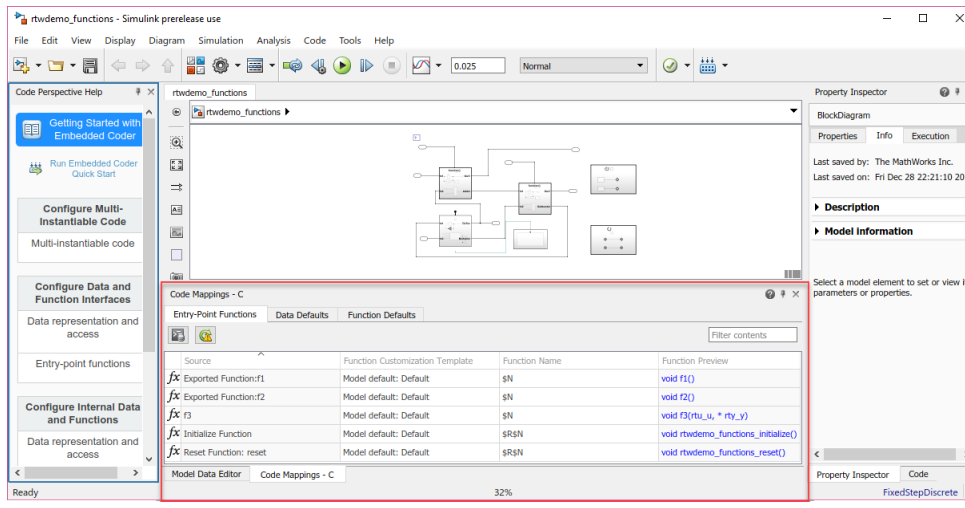
Code Mapping Editor

Associate model data elements and entry-point functions with code definitions

Description

The Code Mappings editor is a graphical interface for configuring model data elements and entry-point functions for code generation. Associate each category of model data element with a specific storage class throughout a model. A storage class defines properties (for example, appearance and location) that the code generator uses when producing code for associated data. Similarly, associate each category of model entry-point functions with a specific function customization template. The templates define how the code generator produces code for associated functions. In the Code Mappings editor, you can override default mappings for specific entry-point functions. Override default mappings for specific data elements by using the **Code** view of the Model Data Editor.

The Code Mappings editor display consists of three tabbed tables: **Entry-Point Functions**, **Data Defaults**, and **Function Defaults**. Use the tables to set code definitions for individual entry-point functions or default code definitions for categories of model data elements and functions. The **Code** section of the Property Inspector shows your selection and whether a memory section is defined for the storage class or function customization template.



Open the Code Mapping Editor

- To prepare a model for code generation, use Embedded Coder Quick Start. Quick Start places your model in Code perspective, which includes the Code Mappings editor.
- In the model window, click the perspective control in the lower-right corner and select **Code**.
- In the model window, select **Code > C/C++ Code > Configure Model in Code Perspective**.
- If you close the Code Mappings editor, the Model Editor window remains in Code perspective mode. To reopen the Code Mappings editor, select **View > Code Mappings**.

Examples

Configure Code Generation for Root Inports and Outports

This example shows how to configure code generation for the root Inport and Outport blocks throughout a model. Applying default configurations can save time, especially for large-scale models that use a significant amount of data. After applying default mappings, you can adjust mappings for individual data elements by using the Model Data Editor.

Set Up Example Environment

1. Copy external code files into your current MATLAB folder.

```
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.h'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.c'));
copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.h'));
```

2. Open model `rtwdemo_roll`.
3. Open code perspective by selecting **Code> C/C++ Code > Configure Model in Code Perspective**.

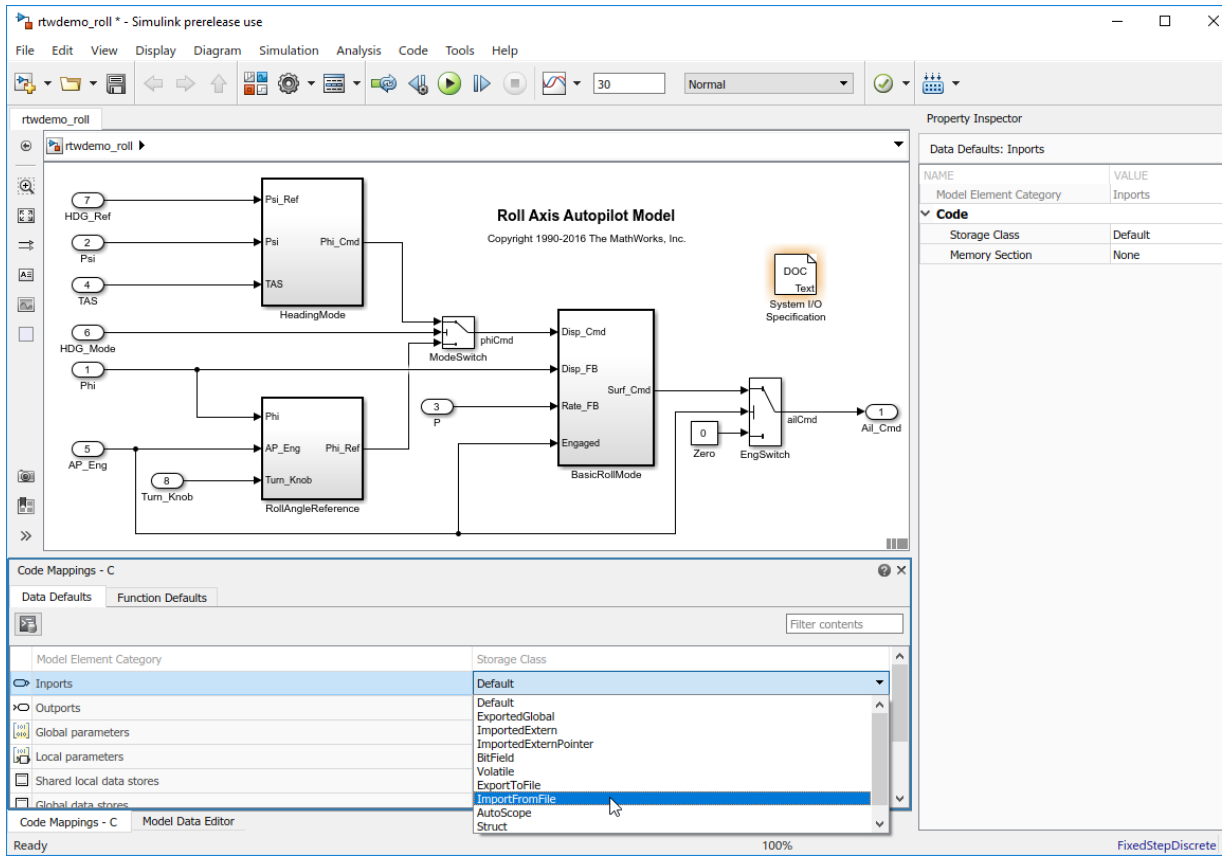
Configure Default Mappings

Configure the code generator to:

- Use header file `roll_input_data.h` to declare the variables representing model Inport blocks.

- Represent variables for model Outport blocks as separate global variables.
- Define output variables in `roll_output_data.c` and declare them in `roll_output_data.h`.

1. Open the Code Mappings editor. Click **Data Defaults** tab if not already selected.
2. Set the storage class for model element category **Inports** to `ImportFromFile`.



3. In the Property Inspector, set **Header File** to `roll_input_data.h`.
4. Set the storage class for model element category **Outports** to `ExportToFile`.
5. In the Property Inspector, set **Header File** to `roll_output_data.h` and **Definition File** to `roll_output_data.c`.

Override Default Mappings

Override the default source location for inport variable `HDG_Mode`. That variable is declared in the external file `roll_heading_mode.h`.

1. Open the Model Data Editor by clicking the **Model Data Editor** tab.
2. Select the `HDG_Mode` row.
3. Set **Storage Class** to `ImportFromFile`.
4. Set **Header File** to `roll_heading_mode.h`.

Configure Model-Wide Parameter Settings

1. Configure the code generator to produce variable names in the code for Inport and Outport blocks that match the variable names in external files `roll_input_data.h` and `roll_heading_mode.h`. Set the model configuration parameter **Global variables** to `$N$M`, removing the `rt` prefix that the code generator applies by default.
2. Include external source files `roll_input_data.c` and `roll_heading_mode.c` in the code generation and build process. Set the model configuration parameter **Source files** to `roll_input_data.c roll_heading_mode.c`.

Generate and Verify Code

Generate code and verify that the code generated for Inport and Output blocks appears as you expected.

- `roll_rtwdemo_roll.h` includes three header files associated with storage classes:

  ```
  #include "roll_output_data.h"
  #include "roll_heading_mode.h"
  #include "roll_input_data.h"
  ```

- `roll_heading_mode.c` includes `roll_heading_mode.h` and defines variable `HDG_Mode`.

  ```
  #include "roll_heading_mode.h"
  boolean_T HDG_Mode;
  ```

- `roll_input_data.c` defines the variables declared in `roll_input_data.h`.

  ```
  #include "roll_input_data.h"

  boolean_T AP_Eng;
  real32_T HDG_Ref;
  ```

```
real32_T Rate_FB;
real32_T Phi;
real32_T Psi;
real32_T TAS;
real32_T Turn_Knob;
```

- `roll_output_data.c` includes this exported data definition:

  ```
  real32_T Ail_Cmd;
  ```

- `roll_output_data.h` includes this exported data declaration:

  ```
  extern real32_T Ail_Cmd;
  ```

Configure Default Function Names for Entry-Point Functions

By default, the code generator uses the identifier naming rule `$R$N` to name entry-point functions. `$R` is the name of the root model. `$N` is the name of the function, for example, `initialize`, `step`, and `terminate`. To integrate generate code with existing external code or to comply with naming standards or guidelines, you can adjust the default naming rule. This example shows how to add the text string `myproj_` as a prefix to `$R$`. Adjusting the default naming rule can save time, especially for multirate models for which the code generator produces a unique `step` function for each rate.

Set Up Example Environment

Open model `rtwdemo_multirate_multitasking` and save a copy to a writable location.

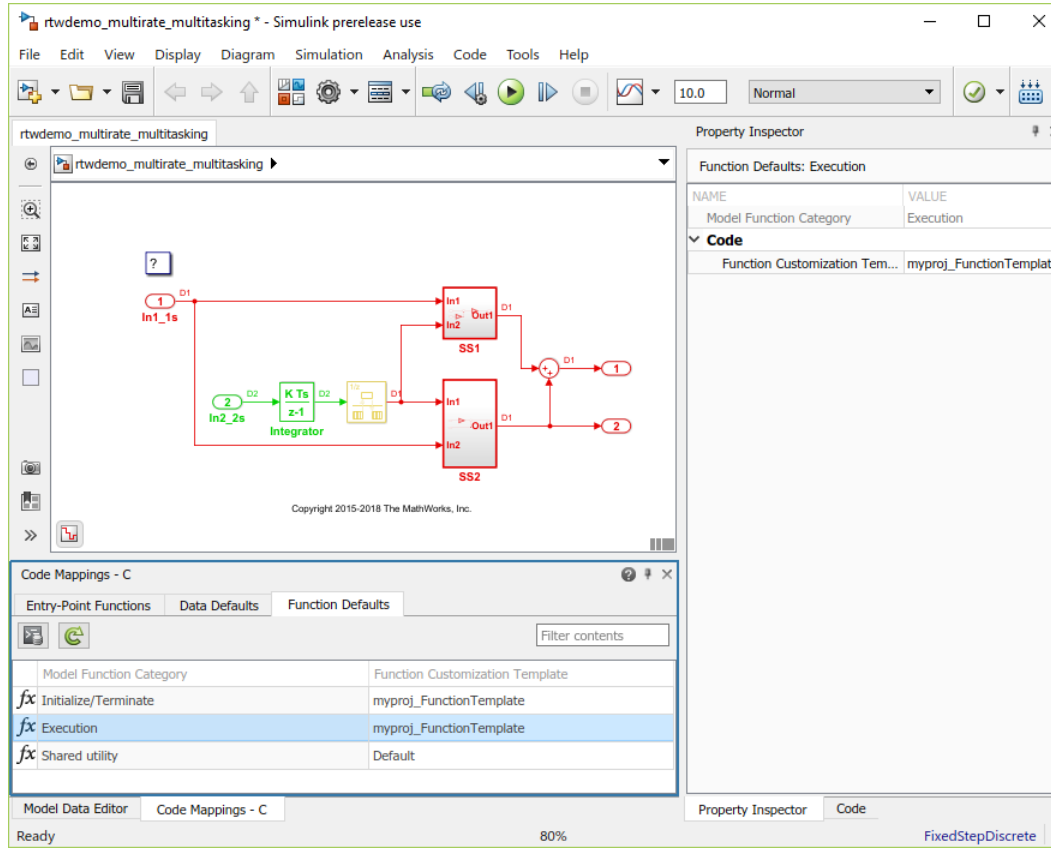
Define Function Naming Rule

Create a function customization template that defines the naming rule `myproj_$R$N`.

1. In the model window, open the Embedded Coder Dictionary by clicking `Code>C/C++ Code>Embedded Coder Dictionary`.
2. Click the **Function Customization Templates** tab.
3. Click **Add**.
4. In the **Name** column of the new table row, name the new template `myproj_FunctionTemplate`.
5. In the **Function Name** column, enter the naming rule `myproj_$R$N`.
6. Close the coder dictionary.

Configure Default Mappings

1. Open code perspective by selecting **Code> C/C++ Code > Configure Model in Code Perspective**.
2. For the **Initialize/Terminate** and **Execution** function categories, change the default function customization template from `Default` to `myproj_FunctionTemplate`.



Generate and Review Code

Generate code and verify the entry-point function names.

```
void myproj_rtwdemo_multirate_multitasking_step0(void) /* Sample time: [1.0s, 0.0s] */
{
  (rtM->Timing.RateInteraction.TID0_1)++;
  if ((rtM->Timing.RateInteraction.TID0_1) > 1) {
```

```
       rtM->Timing.RateInteraction.TID0_1 = 0;
  }

  if (rtM->Timing.RateInteraction.TID0_1 == 1) {
      rtDW.RateTransition = rtDW.RateTransition_Buffer0;
  }
      rtY.Out2 = 2.0 * rtDW.RateTransition + rtU.In1_1s;
      rtY.Out1 = (3.0 * rtDW.RateTransition + rtU.In1_1s) * 5.0 + rtY.Out2;
}

/* Model step function for TID1 */
void myproj_rtwdemo_multirate_multitasking_step1(void) /* Sample time: [2.0s, 0.0s] */
{
  rtDW.RateTransition_Buffer0 = rtDW.Integrator_DSTATE;
  rtDW.Integrator_DSTATE += 2.0 * rtU.In2_2s;
}

void myproj_rtwdemo_multirate_multitasking_initialize(void)
{
  /* (no initialization code required) */
}

void myproj_rtwdemo_multirate_multitasking_terminate(void)
{
  /* (no terminate code required) */
}
```

Customize Individual Entry-Point Functions

In general, you can customize the names of entry-point functions and the arguments of execution functions, such as step functions and Simulink functions, for a model. This example shows how to customize the entry-point functions for the model `rtwdemo_roll`.

Set up the environment

1. Copy external code files into your current MATLAB folder.

   ```
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.c'));
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_input_data.h'));
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.c'));
   copyfile(fullfile(matlabroot,'toolbox','rtw','rtwdemos','roll_heading_mode.h'));
   ```

2. Open the model `rtwdemo_roll`.
3. Enter the Code perspective by selecting **Code> C/C++ Code > Configure Model in Code Perspective**.

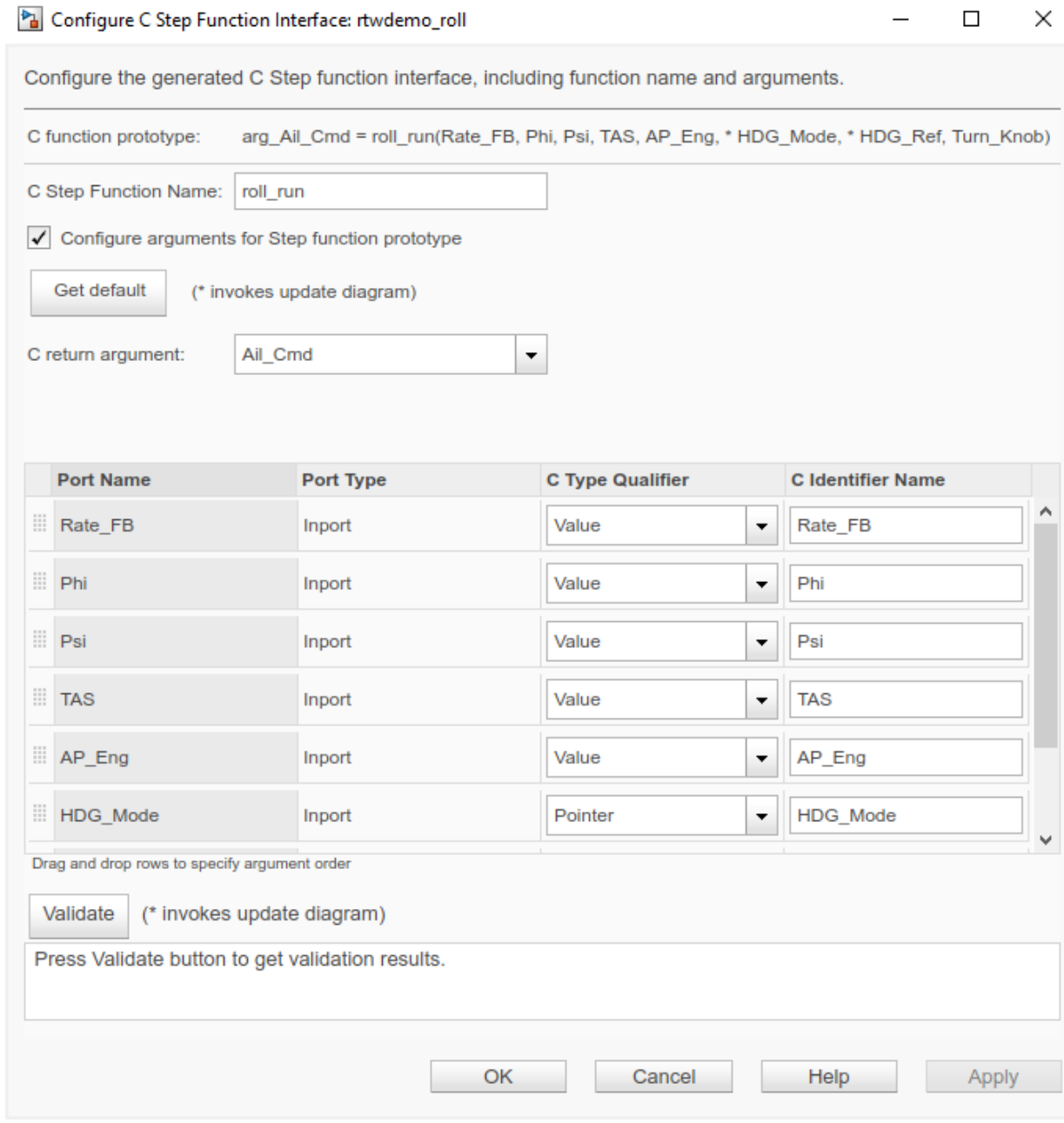
Customize entry-point functions

1. Open the Code Mappings editor and select the **Entry-Point Functions** tab.
2. Customize the function name of an entry-point function using one of these methods:

- Under the **Function Name** column, directly edit the name of the function.
- Under the **Function Preview** column, click the prototype hyperlink of the function to open a configuration dialog box. In the `C Initialize Function Name` field edit the function name.

For this example, change the function name to `roll_run`.

3. Customize the arguments of execution entry-point functions. For this example, customize the `step` function. Open the configuration dialog box for the `step` function by clicking the prototype hyperlink located under the **Function Preview** column.
4. Select the **Configure arguments for Step function prototype** check box. Click **Get Default** to open a table that displays the default configurations for the arguments.
5. Customize the arguments with the following changes:
   - From the **C return argument** drop down list, select `Ail_Cmd`.
   - For each port, in the **C Identifier Name** field, remove the `arg_` prefix from their default names.
   - For the `HDG_Mode` Inport, from the **C Type Qualifer** drop down list, select `Pointer`. In the **C Identifier Name** field change the name to `HDG_Ref`
6. Click **Apply** and verify that the function prototype reflects the changes.


Configure C Step Function Interface: rtwdemo_roll
Configure the generated C Step function interface, including function name and arguments.
C function prototype: arg_Ail_Cmd = roll_run(Rate_FB, Phi, Psi, TAS, AP_Eng, * HDG_Mode, * HDG_Ref, Turn_Knob)
C Step Function Name: roll_run
Configure arguments for Step function prototype
Get default
(* invokes update diagram)
C return argument: Ail_Cmd
Port Name
Port Type
C Type Qualifier
C Identifier Name
Rate_FB Inport Value Rate_FB
Phi Inport Value Phi
Psi Inport Value Psi
TAS Inport Value TAS
AP_Eng Inport Value AP_Eng
HDG_Mode Inport Pointer HDG_Mode
Drag and drop rows to specify argument order
Validate
(* invokes update diagram)
Press Validate button to get validation results.
OK
Cancel
Help
Apply

7 Validate the changes by clicking **Validate**.

8 Click **OK** to exit the dialog box.

Generate and Verify Code

1 Generate the code and navigate to the Code view.

2 Verify the updates in the generated C file, `rtwdemo_roll.c`. Use the **Search** field to find the updated `step` function (`roll_run`) as an element in the list.

3 Select the `step` function to verify its prototype.

```
real32_T roll_run(real32_T Phi, real32_T Psi, real32_T Rate_FB, real32_T TAS,
                  boolean_T AP_Eng, boolean_T *HDG_Mode, real32_T *HDG_Ref,
                  real32_T Turn_Knob)
```

Parameters

Entry-Point Functions

`Source` — Type of entry-point function

character vector

Identifies the type of entry-point function. For rate-based models, this property also provides the sample rate of the `step` functions.

`Function Customization Template` — Code definition for function

character vector

Definition (specification) that the code generator uses to determine properties, such as appearance and location, for code that it produces for a model function.

`Function Name` — Name for function

character vector

Name that the code generator uses for a model function. For the step function (base rate step function for rate based models), the **Function Name** table cell provides access to a button that opens the Configure C Step Function Interface dialog box. Click the name, click the vertical dots, and click **Configure Prototype**. Use the Configure C Step Function Interface dialog box to customize the entire step function interface.

Data Defaults

`Model Element Category` — Category of model data element

character vector

Names a category of Simulink model elements. The storage class that you set for a category applies to elements in that category throughout the model.

Model Element Category	Description
Inports	Root-level input ports of a model.
Outports	Root-level output ports of a model.
Global parameters	Parameters that are defined in the base workspace or in a data dictionary. Multiple models in an application can use these parameters.
Local parameters	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
Parameter arguments	Block parameters in the model workspace that are configured as model arguments. These parameters are exposed at the model block to allow each model instance to provide its own value. To specify a parameter as a model argument, select the **Argument** check box on the **Parameters** tab in the Model Data Editor.
Shared local data stores	Data Store Memory blocks with the block parameter **Share across model instances** set. These data stores are accessible only in the model where they are defined. The data store value is shared across instances of the model.
Global data stores	Data stores that are defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores.
Internal data	Local data, such as data stores, discrete block states, block output signals, and zero-crossing signals.
Constants	Constant-value block output and constant parameters in a model.

`Storage Class` — Code definition for model data elements
character vector

Definition (specification) that the code generator uses to determine properties, such as appearance and location, for code that it produces for model data elements.

Function Defaults

`Model Function Category` — Category of model functions
character vector

Names a category of Simulink model functions. The function customization template that you set for a category applies to functions in that category throughout the model.

Model Function Category	Description
Initialize/Terminate	Entry-point functions for initialization and termination
Execution	Entry-point functions for initiating execution and resets
Shared utility	Shared utility functions

`Function Customization Template` — Code definition for functions
character vector

Definition (specification) that the code generator uses to determine properties, such as appearance and location, for code that it produces for model functions.

See Also

Embedded Coder Dictionary

Topics

“Control Data and Function Interface in Generated Code”
“Configure Default C Code Generation for Categories of Model Data and Functions”
“Override Default Naming for Individual C Entry-Point Functions”
“Override Default C Step Function Interface”
“Configure Code Generation for Model Entry-Point Functions”

Introduced in R2018a

Code Replacement Tool

Create, modify, and validate content of code replacement libraries

Description

The Code Replacement Tool is a graphical interface that you can use to create and manage custom code replacement libraries. You can create, import, manipulate, and validate the code replacement tables in a library. The tool also generates the customization file to register a code replacement library with the code generator. If you specify a table name when you open the tool, the tool displays only the contents of that table.

The tool display consists of three panes that show table and table entry information:

- Left pane lists code replacement tables.
- Middle pane lists available tables or, if you select a table in the left pane, the table entries that are in that table.
- Right pane lists table or table entry details. If you select a table, the right pane shows table properties: the table name, which you can modify, the table version, and the total number of entries in the table. If select a table entry, the right pane shows mapping and build information for that entry.

Open the Code Replacement Tool

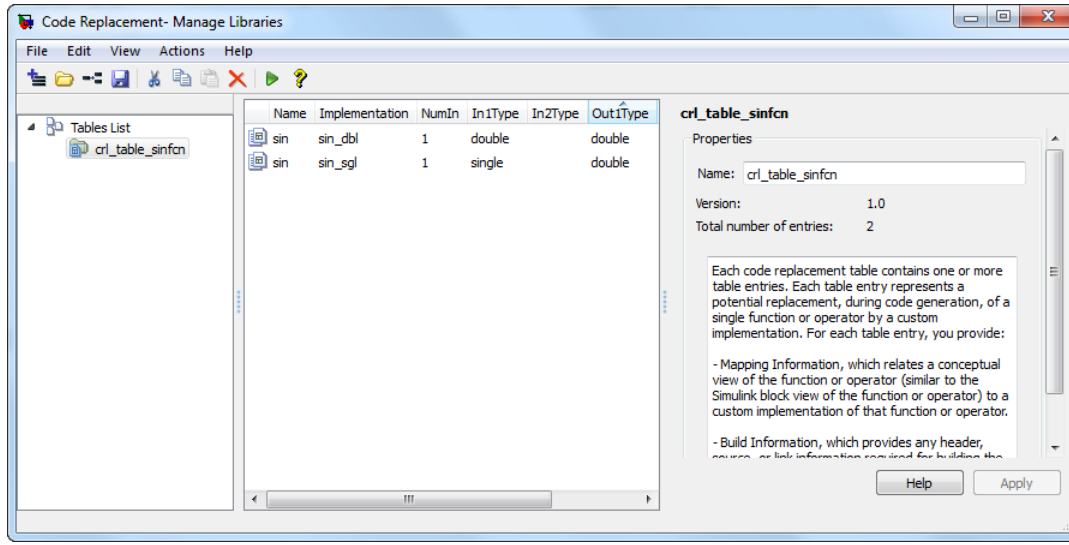
At the command prompt, type `crtool`.

Examples

Open an Existing Table in the Tool

This example shows how to open a code replacement table, `crl_table_sinfcn`, in the Code Replacement Tool.

```
crtool('crl_table_sinfcn')
```



- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Quick Start Code Replacement Library Development - Simulink®”

Parameters

Entry Summary Information (Center Pane)

Name — Name of table entry (read-only)
character vector

Conceptual name of the function or operation being replaced. Can name a math operation, function, BLAS operation, CBLAS operation, net slope fixed-point operation, semaphore or mutex entry, or customization entry.

`Implementation` — Name of replacement function
character vector

Name of the implementation (replacement) function.

`NumIn` — Number of input arguments (read-only)
scalar integer

Number of input arguments.

`In`*`n`*`Type` — Data type of conceptual input argument
character vector

Data type of a conceptual input argument.

`Out`*`n`*`Type` — Data type of conceptual output argument
character vector

Data type of a conceptual output argument.

`Priority` — Entry match priority
100 (default) | integer, ranging from 0 to 100

The entry match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Entry Mapping Information (Right Pane)

`Function/Operation` — Name of table entry
character vector

Conceptual name of the function or operation being replaced. Can name a math operation, function, BLAS operation, CBLAS operation, net slope fixed-point operation, semaphore or mutex entry, or customization entry.

`Algorithm` — Computation or approximation algorithm
unspecified (default) | options vary depending on function or operation

Computation or approximation algorithm configured for a function or operation being replaced. For example, you can configure:

- The Reciprocal Sqrt block to use the `Newton-Raphson` computation method.
- The Trigonometric Function block, with **Function** set to `sin`, `cos`, or `sincos`, to use the approximation method `CORDIC`.
- An addition or subtraction operation, to use the cast-before-operation or cast-after-operation algorithm.

`Conceptual arguments` — Conceptual argument names
yn | un

Names of input and output arguments of function or operation being replaced. Conceptual arguments observe naming conventions (`y1`, `u1`, `u2`, ...) and data types familiar to the code generator.

`Data type (conceptual)` — Conceptual argument data type
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | void | logical | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0)

Data type of a selected input or output argument of the function or operation being replaced. Conceptual arguments observe data types familiar to the code generator.

`Complex (conceptual)` — Conceptual argument complexity
cleared (default) | selected

Whether the selected input or output argument of the function or operation being replaced is real or complex.

`Argument type` — Conceptual argument type
scalar (default) | matrix

Whether the selected input or output argument of the function or operation being replaced is a scalar value or a matrix. If you select `Matrix`, parameters for specifying range dimensions, and for replacement of MATLAB code, array layout appear.

`Lower range` — Lower range of matrix dimensions
Column-major (default) | Row-major| | Column-and-Row

Vector that specifies the lower range of the matrix dimensions.

`Upper range` — Upper range of matrix dimensions
[2 2] (default)

Vector that specifies the upper range of the matrix dimensions.

`Array layout supported by entry` — Layout for array storage
Column-major (default) | Row-major | Column-and-Row

Order in which array elements are stored in memory. Row-major layout can improve performance for certain algorithms and ease integration with external code or data that uses the row-major layout.

`Make conceptual and implementation argument types the same` — Data type consistency
selected (default) | cleared

Whether you want the data types for your implementation arguments to be the same as the conceptual argument types. For example, most ANSI-C functions operate on and return `double` data. Clear the check box if want to map the conceptual representation of a function or operation to an implementation representation that specifies an argument and return value. For example, clear the check box to map the conceptual representation of the function `sin` to an implementation representation that specifies an argument and return value of type `single` (`single sin(single)`).

`Name` — Name of replacement function
character vector

Name of the replacement function.

`C++ namespace` — Namespace of replacement function
character vector

Namespace of the replacement function.

`Function returns void` — Function returns void
selected (default) | cleared

Whether your implementation function returns `void`.

`Function arguments` — Replacement argument names
yn | un

Names of input and output arguments of your replacement function.

`Data type (replacement)` — Replacement argument data type
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | void | integer | size_t | long | ulong | long long | ulong long | char

Data type of a selected input or output argument of your replacement function.

`I/O type` — Replacement argument I/O type
OUTPUT | INPUT

Whether a selected argument of your replacement function is an input or output argument.

`Const` — Const replacement argument
cleared (default) | selected

Whether to apply the `const` type qualifier to a selected argument of your replacement function.

`Pointer` — Pointer replacement argument
cleared (default) | selected

Whether a selected argument of your replacement function is a pointer.

`Complex (replacement)` — Replacement argument complexity
cleared (default) | selected

Whether the selected input or output argument of the replacement function is real or complex.

`Integer saturation mode` — Saturation mode
unspecified Saturation (default) | wrap on overflow | saturate on overflow

Saturation mode supported by the replacement function.

`Rounding modes` — Rounding modes
unspecified rounding (default) | floor | ceil | zero | nearest | MATLAB nearest | simplest | conv

Rounding modes supported by the replacement function.

`Allow expressions as inputs` — Expressions as inputs
selected (default) | cleared

Whether your replacement function accepts expression inputs. If you select the parameter, the code generator integrates an expression input into the generated code rather than inserting a temporary variable in place of the expression input.

`Function modifies internal or global state` — State modification
cleared (default) | selected

Whether your replacement function modifies variables representing internal or global state.

Entry Build Information (Right Pane)

`Implementation header file` — Header file for replacement function
character vector

Header file for the replacement function (for example, `my_rep_func.h`).

`Implementation source file` — Source file for replacement function
character vector

Source file for the replacement function (for example, `my_rep_func.c`).

`Additional header files/include paths` — Names and paths of additional header files
character vector

Names and paths of additional header files to include for the replacement function (for example, `support_files.h` and `matlab\customization\mylib\include`).

`Additional source files/ paths` — Names and paths of additional source files
character vector

Names and paths of additional source files to include for the replacement function (for example, `support_files.c` and `matlab\customization\mylib\src`).

`Additional object files/ paths` — Names and paths of link object files
character vector

Names and paths of link object files to use for the replacement function (for example, `support_files.o` and `matlab\customization\mylib\bin`).

`Additional link flags` — Link flags to use
character vector

Link flags to use for the replacement function (for example, `-MD -Gy`).

`Additional compile flags` — Compile flags to use
character vector

Compile flags to use for the replacement function (for example, `-Zi -Wall`).

`Copy files to build directory` — Copy files to build folder
cleared (default) | selected

Whether the code generator copies files from external folders to the build folder before starting the build process.

Programmatic Use

`crtool(table)` opens the Code Replacement Tool and displays the contents of `table`, where `table` is a character vector that names a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

See Also

Topics

“What Is Code Replacement?”
“What Is Code Replacement Customization?”
“Quick Start Code Replacement Library Development - Simulink®”

Introduced in R2014b

Code Replacement Viewer

Explore content of code replacement libraries

Description

The Code Replacement Viewer displays the content of code replacement libraries and tables. You can use this tool to explore and choose a code replacement library or to view a predefined code replacement table. If you develop a custom code replacement library, you can use this viewer to verify table entries for the following properties:

- Argument order is correct.
- Conceptual argument names match code generator naming conventions.
- Implementation argument names are correct.
- Header or source file specification is not missing.
- I/O types are correct.
- Relative priority of entries is correct (highest priority is 0, and lowest priority is 100).
- Saturation or rounding mode specifications are not missing.

If you specify a library name when you open the viewer, the viewer displays the code replacement tables for that library. If you specify a table name when you open the viewer, the viewer displays the function and operator code replacement entries for that table. The viewer can only display code replacement tables that are defined. For more information on creating code replacement tables, see “Define Code Replacement Mappings”.

Abbreviated Entry Information

In the middle pane, the viewer displays entries that are in the selected code replacement table, along with abbreviated information for each entry.

Field	Description
Name	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`).
Implementation	Name of the implementation function, which can match or differ from **Name**.

Field	Description
NumIn	Number of input arguments.
In1Type	Data type of the first conceptual input argument.
In2Type	Data type of the second conceptual input argument.
OutType	Data type of the conceptual output argument.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
UsageCount	Not used.

Detailed Entry Information

In the middle pane, when you select an entry, the viewer displays entry details.

Field	Description
Description	Text description of the table entry (can be empty).
Key	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`), and the number of conceptual input arguments.
Implementation	Name of the implementation function, and the number of implementation input arguments.
Implementation type	Type of implementation: `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Saturation mode	Saturation mode that the implementation function supports. One of:
	`RTW_SATURATE_ON_OVERFLOW`
	`RTW_WRAP_ON_OVERFLOW`
	`RTW_SATURATE_UNSPECIFIED`

Field	Description

Rounding modes	Rounding modes that the implementation function supports. One or more of:
	`RTW_ROUND_FLOOR`
	`RTW_ROUND_CEILING`
	`RTW_ROUND_ZERO`
	`RTW_ROUND_NEAREST`
	`RTW_ROUND_NEAREST_ML`
	`RTW_ROUND_SIMPLEST`
	`RTW_ROUND_CONV`
	`RTW_ROUND_UNSPECIFIED`

GenCallback file	Not used.
Implementation header	Name of the header file that declares the implementation function.
Implementation source	Name of the implementation source file.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
Total Usage Count	Not used.
Entry class	Class from which the current table entry is instantiated.
Conceptual arguments	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), and data type for each conceptual argument.
Implementation	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), data type, and alignment requirement for each implementation argument.

Fixed-Point Entry Information

When you select an operator entry that specifies net slope fixed-point parameters, the viewer displays fixed-point information.

Field	Description
Net slope adjustment factor F	Slope adjustment factor (F) part of the net slope factor, $F2^E$, for net slope table entries. You use this factor with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Net fixed exponent E	Fixed exponent (E) part of the net slope factor, $F2^E$, for net slope table entries. You use this fixed exponent with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Slopes must be the same	Indicates whether code replacement request processing must check that the slopes on arguments (input and output) are equal. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.
Must have zero net bias	Indicates whether code replacement request processing must check that the net bias on arguments is zero. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.

Open the Code Replacement Viewer

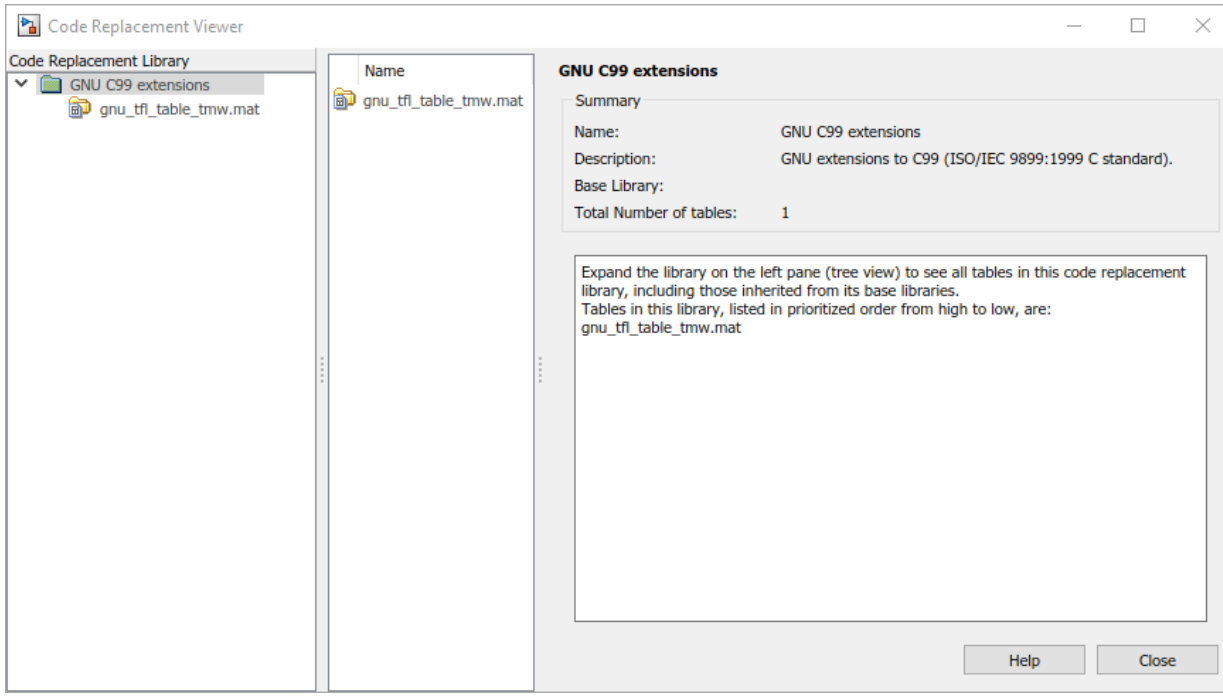
Open from the MATLAB command prompt using `crviewer`.

Examples

Display Contents of Code Replacement Library

This example opens the registered code replacement library `GNU C99 extensions`.

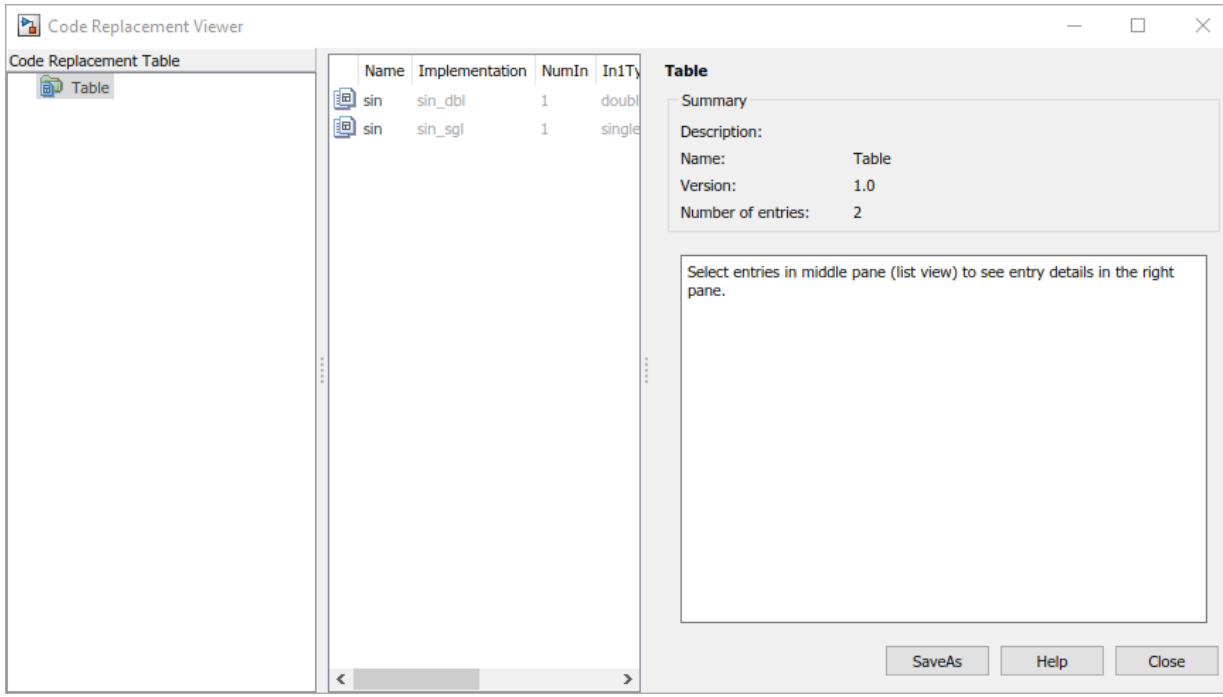
```
crviewer('GNU C99 extensions')
```



Display Contents of Code Replacement Table

This example opens a predefined code replacement table `crl_table_sinfcn`. To learn how to create this example table, see “Define Code Replacement Mappings”.

```
crviewer(crl_table_sinfcn)
```



- “Choose a Code Replacement Library”
- “Verify Code Replacements”

Programmatic Use

`crviewer('library')` opens the Code Replacement Viewer and displays the contents of `library`, where `library` is a character vector that names a registered code replacement library.

`crviewer(table)` opens the Code Replacement Viewer and displays the contents of a predefined `table`, where `table` is a MATLAB file that defines code replacement tables. The table must be user predefined and the file must be in the current folder or on the MATLAB path.

See Also

Topics

“Choose a Code Replacement Library”
“Verify Code Replacements”
“What Is Code Replacement?”
“What Is Code Replacement Customization?”
“Code Replacement Libraries”
“Code Replacement Terminology”

Introduced in R2014b

17

C/C++ Functions That Support Symbolic Dimensions for Simulink Function Blocks

ssSetSymbolicDimsSupport

Specify whether an S-function supports symbolic dimensions

Languages

C, C++

Syntax

```
void ssSetSymbolicDimsSupport(SimStruct *S, const boolean_T val)
```

Arguments

`S`

SimStruct representing an S-Function block.

`val`

Boolean value corresponding to whether the S-Function block supports symbolic dimensions.

Returns

This function does not return a value.

Example

Call this function from inside the `mdlInitializeSizes` function. For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”

Introduced in R2016a

mdlSetInputPortSymbolicDimensions

Specify symbolic dimensions of an input port and how those dimension propagate forward

Languages

C, C++

Syntax

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_INPUT_PORT_SYMBOLIC_DIMENSIONS

static void mdlSetInputPortSymbolicDimensions(SimStruct *S, int_T portIndex,
 SymbDimsId symbDimsId)

{
}
#endif
```

Arguments

`S`

SimStruct representing an S-Function block.

`portIndex`

Index of an input port.

`symbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

mdlSetOutputPortSymbolicDimensions

Specify symbolic dimensions of an output port and how those dimension propagate backward

Languages

C, C++

Syntax

```
#if defined(MATLAB_MEX_FILE)
#define MDL_SET_OUTPUT_PORT_SYMBOLIC_DIMENSIONS

static void mdlSetOutputPortSymbolicDimensions(SimStruct *S, int_T portIndex,
 SymbDimsId symbDimsId)

{
}
#endif
```

Arguments

`S`

SimStruct representing an S-Function block.

`portIndex`

Index of an output port.

`symbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

Call this function from inside the `mdlInitializeSizes` function. For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssRegisterSymbolicDimsExpr

Create `SymbDimsId` from expression string (`aExpr`)

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsExpr(SimStruct *S, const char_T* aExpr)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aExpr`

Expression string that forms a valid syntax in C.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example creates a `SymbDimsId` for the expression string `[ F / C , D * (B-3)]`.

```
const SymbDimsId outputDimsId =
       ssRegisterSymbolicDimsExpr(S, "[ F / C , D * (B-3)]");
```

Introduced in R2016a

ssRegisterSymbolicDims

Create `SymbDimsId` from array of `SymDimsId`s

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDims(SimStruct *S, const SymbDimsId* aDimsVec,
 const size_t aNumDims)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aDimsVec`

Array of `SymDimsId`s

`aNumDims`

Size of `SymDimsId` array

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssRegisterSymbolicDimsString

Create `SymbDimsId` from identifier string

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsString(SimStruct *S, const char_T* aString)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aString`

Identifier string

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example creates a `SymbDimsId` for the string `"B"`.

```
 const SymbDimsId symbolId = ssRegisterSymbolicDimsString(S, "B");
```

Introduced in R2016a

ssRegisterSymbolicDimsIntValue

Create `SymbDimsId` from integer value

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsIntValue(SimStruct *S, const int_T aIntValue)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aIntValue`

Dimensions value

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example creates a `SymbDimsId` for the integer `2`.

```
 const SymbDimsId symbolId = ssRegisterSymbolicDimsIntValue(S, 2);
```

Introduced in R2016a

ssRegisterSymbolicDimsPlus

Create `SymbDimsId` by adding two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsPlus(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example shows how to add the `SymbDimsId` `symbDims` to `symbolId`, and then sets the result equal to a new `SymbDimsId` called `outputDimsId`.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsPlus(S, symbDimsId, symbolId);
```

Introduced in R2016a

ssRegisterSymbolicDimsMinus

Create `SymbDimsId` by subtracting two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsMinus(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

For an example of how to use this function to configure an S-function that supports forward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssRegisterSymbolicDimsMultiply

Create `SymbDimsId` by multiplying two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsMultiply(SimStruct *S, const SymbDimsId aLHS,
 const SymbDimsId aRHS)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

aRHS

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example shows how to multiply the `SymbDimsId`s `symbDimsId` and `symbolId`. It sets the result equal to a new `SymbDimsId` called `outputDimsId`.

```
const SymbDimsId outputDimsId =
       ssRegisterSymbolicDimsMultiply(S, symbDimsId, symbolId);
```

Introduced in R2016a

ssRegisterSymbolicDimsDivide

Create `SymbDimsId` by dividing two symbolic dimensions

Languages

C, C++

Syntax

```
SymbDimsId ssRegisterSymbolicDimsDivide(SimStruct *S, const SymbDimsId aLHS,
  const SymbDimsId aRHS)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aLHS`

Unique integer value corresponding to a symbolic dimension specification.

`aRHS`

Unique integer value corresponding to a symbolic dimension specification.

Returns

A unique `SymbDimsId` (integer value) that represents a symbolic dimension specification.

Example

This example shows how to divide the `SymbDimsId` `symbDimsId` by `symbolId`. It sets the result equal to a new `SymbDimsId` called `outputDimsId`.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsDivide(S, symbDimsId, symbolId);
```

Introduced in R2016a

ssGetNumSymbolicDims

Get the number of dimensions for `SymbDimsId`

Languages

C, C++

Syntax

```
size_t  ssGetNumSymbolicDims(SimStruct *S, const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

The number of dimensions for a `SymbDimsId`.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssGetSymbolicDim

Get `SymbDimsId` from array of `SymbDimsId`s

Languages

C, C++

Syntax

```
SymbDimsId ssGetSymbolicDim(SimStruct *S, const SymbDimsId aSymbDimsId,
 const int_T aDimsIdx)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aSymbDimsId`

Unique integer corresponding to a symbolic dimension specification.

`aDimsIdx`

Array index

Returns

A unique `SymbDimsId`.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetInputPortSymbolicDimsId

Set precompiled `SymbDimsId` of input port

Languages

C, C++

Syntax

```
void ssSetInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Array index

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

You can call this function from inside the `mdlInitializeSizes` function. For an input port with an index of `0`, this example shows how to set the precompiled `SymbDimsId` equal to `inputDimsId`.

```
const SymbDimsId inputDimsId = ssRegisterSymbolicDimsExpr(S, "[A+3, B-2]");
    ssSetInputPortSymbolicDimsId(S, 0, inputDimsId);
```

Introduced in R2016a

ssGetCompInputPortSymbolicDimsId

Get compiled `SymbDimsId` of input port

Languages

C, C++

Syntax

```
SymbDimsId  ssGetCompInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port

Returns

`SymbDimsId` corresponding to symbolic dimensions of an input port.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetCompInputPortSymbolicDimsId

Set compiled `SymbDimsId` of an input port

Languages

C, C++

Syntax

```
void ssSetCompInputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
  const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

For examples of how to use this function to configure S-functions that support forward propagation of symbolic dimensions and forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetOutputPortSymbolicDimsId

Set precompiled `SymbDimsId` of an output port.

Languages

C, C++

Syntax

```
void ssSetOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

You can call this function from inside the `mdlInitializeSizes` function. For an output port with an index of `0`, this example shows how to set the precompiled `SymbDimsId` equal to `outputDimsId`.

```
const SymbDimsId outputDimsId =
        ssRegisterSymbolicDimsExpr(S, "[ F / C , D * (B-3)]");
    ssSetOutputPortSymbolicDimsId(S, 0, outputDimsId);
```

Introduced in R2016a

ssGetCompOutputPortSymbolicDimsId

Get compiled `SymbDimsId` of output port

Languages

C, C++

Syntax

```
SymbDimsId ssGetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an output port.

Returns

`SymbDimsId` corresponding to symbolic dimensions of an output port.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetCompOutputPortSymbolicDimsId

Set compiled `SymbDimsId` of output port

Languages

C, C++

Syntax

```
void ssSetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-function block.

`aPortIdx`

Index of an output port.

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Example

For an example of how to use this function to configure an S-function that supports forward and backward propagation of symbolic dimensions, see “Configure Dimension Variants for S-Function Blocks”.

Introduced in R2016a

ssSetCompDWorkSymbolicDimsId

Set compiled `SymbDimsId` of an index of a block's data type work (`DWork`) vector

Languages

C, C++

Syntax

```
void ssSetCompOutputPortSymbolicDimsId(SimStruct *S, const int_T aPortIdx,
 const SymbDimsId aSymbDimsId)
```

Arguments

`S`

SimStruct representing an S-Function block.

`aPortIdx`

Index of an input port

`aSymbDimsId`

Unique integer value corresponding to a symbolic dimension specification.

Returns

This function does not return a value.

Introduced in R2016a