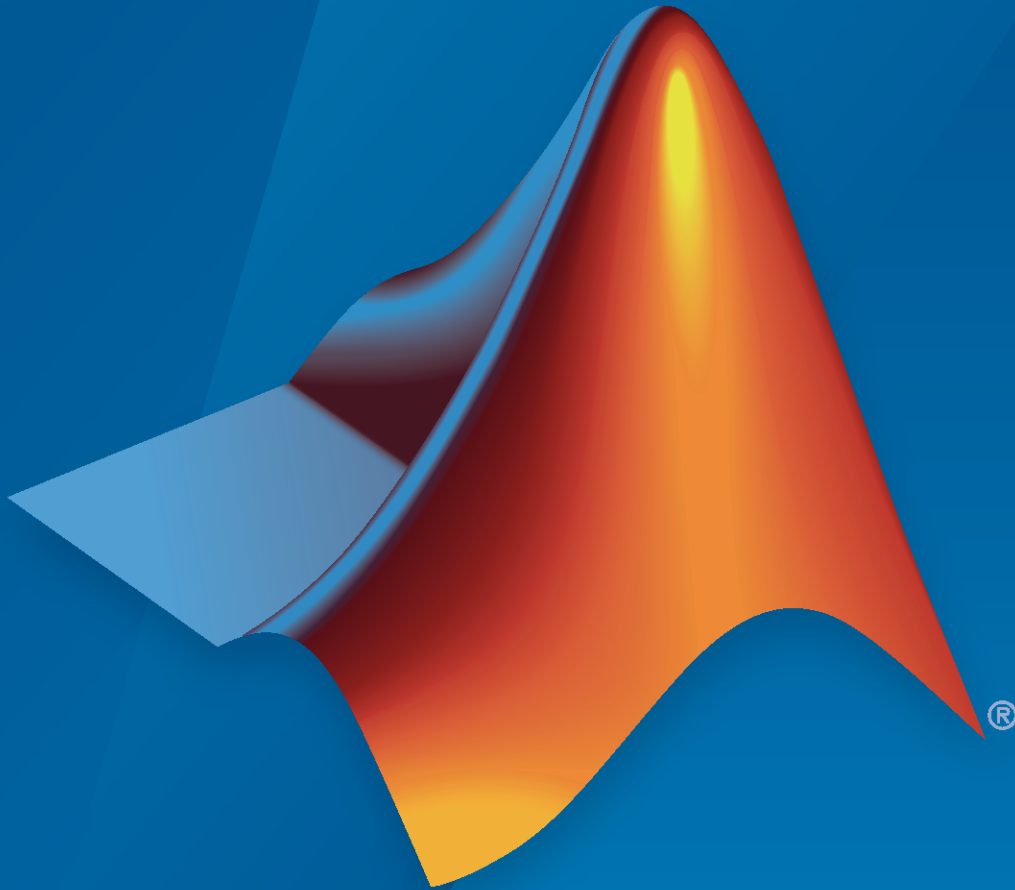


Embedded Coder®

Reference



MATLAB® & SIMULINK®

R2026a



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

© COPYRIGHT 2011–2026 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

April 2011	Online only
September 2011	Online only
March 2012	Online only
September 2012	Online only
March 2013	Online only
September 2013	Online only
March 2014	Online only
October 2014	Online only
March 2015	Online only
September 2015	Online only
October 2015	Online only
March 2016	Online only
September 2016	Online only
March 2017	Online only
September 2017	Online only
March 2018	Online only
September 2018	Online only
March 2019	Online only
September 2019	Online only
March 2020	Online only
September 2020	Online only
March 2021	Online only
September 2021	Online only
March 2022	Online only
September 2022	Online only
March 2023	Online only
September 2023	Online only
March 2024	Online only
September 2024	Online only
March 2025	Online only
September 2025	Online only
March 2026	Online only

New for Version 6.0 (Release 2011a)
Revised for Version 6.1 (Release 2011b)
Revised for Version 6.2 (Release 2012a)
Revised for Version 6.3 (Release 2012b)
Revised for Version 6.4 (Release 2013a)
Revised for Version 6.5 (Release 2013b)
Revised for Version 6.6 (Release 2014a)
Revised for Version 6.7 (Release 2014b)
Revised for Version 6.8 (Release 2015a)
Revised for Version 6.9 (Release 2015b)
Rereleased for Version 6.8.1 (Release 2015aSP1)
Revised for Version 6.10 (Release 2016a)
Revised for Version 6.11 (Release 2016b)
Revised for Version 6.12 (Release 2017a)
Revised for Version 6.13 (Release 2017b)
Revised for Version 7.0 (Release 2018a)
Revised for Version 7.1 (Release 2018b)
Revised for Version 7.2 (Release 2019a)
Revised for Version 7.3 (Release 2019b)
Revised for Version 7.4 (Release 2020a)
Revised for Version 7.5 (Release 2020b)
Revised for Version 7.6 (Release 2021a)
Revised for Version 7.7 (Release 2021b)
Revised for Version 7.8 (Release 2022a)
Revised for Version 7.9 (Release 2022b)
Revised for Version 7.10 (Release 2023a)
Revised for Version 23.2 (R2023b)
Revised for Version 24.1 (R2024a)
Revised for Version 24.2 (R2024b)
Revised for Version 25.1 (R2025a)
Rereleased for Version 25.2 (R2025b)
Revised for Version 26.1 (R2026a)

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.

1	Embedded Coder Functions	
2	Simulink Coder Functions	
3	Embedded Coder Blocks	
4	Simulink Coder Blocks	
5	Embedded Coder Parameters: Advanced Parameters	
	Use only existing shared code	5-2
	Description	5-2
	Settings	5-2
	Dependency	5-2
	Command-Line Information	5-2
	Recommended Settings	5-2
	Use Embedded Coder Features	5-4
	Description	5-4
	Settings	5-4
	Dependencies	5-4
	Command-Line Information	5-4
	Feature set for selected hardware board	5-5
	Settings	5-5

Code Generation Parameters: AUTOSAR

6

Model Configuration Parameters: Code Generation AUTOSAR	6-2
Code Generation: AUTOSAR Code Generation Options Tab Overview	6-3
Configuration	6-3
To get help on an option	6-3
Tip	6-3

Code Generation Parameters: Code Placement

7

Model Configuration Parameters: Code Generation Code Placement	7-2
Code Generation: Code Placement Tab Overview	7-3
Configuration	7-3
To get help on an option	7-3

Code Generation Parameters: Code Style

8

Model Configuration Parameters: Code Style	8-2
Code Generation: Code Style Tab Overview	8-4
Configuration	8-4
To get help on an option	8-4

Code Generation Parameters: Code Generation

9

Model Configuration Parameters: Code Generation	9-2
Set Objectives	9-12
Description	9-12
Dependency	9-12
Set Objectives — Code Generation Advisor Dialog Box	9-13
Description	9-13
Settings	9-13
Dependency	9-14
Command-Line Information	9-14

10	Code Generation Parameters: Optimization	
	Model Configuration Parameters: Code Generation Optimization	10-2
11	Code Generation Parameters: Comments	
	Model Configuration Parameters: Comments	11-2
	Auto generated comments	11-2
	Custom comments	11-2
12	Code Generation Parameters: Report	
	Model Configuration Parameters: Code Generation Report	12-2
	Configure	12-10
	Description	12-10
	Dependency	12-10
13	Code Generation Parameters: Custom Code	
	Model Configuration Parameters: Code Generation Custom Code	13-2
14	Model Configuration Parameters: Code Generation Interface	
	Model Configuration Parameters: Code Generation Interface	14-2
15	Code Generation Parameters: Identifiers	
	Model Configuration Parameters: Code Generation Identifiers	15-2

Model Configuration Parameters: Code Generation Data Type Replacement	16-2
Configure Simulink Coder Data Type Replacements Programmatically	16-3
Code Generation: Data Type Replacement Tab	16-5
Configuration	16-5
To get help on an option	16-5

Memory Sections Parameters on the Code Generation Pane

Code Generation: Memory Sections Tab Overview	17-2
Configuration	17-2
To get help on an option	17-2
Package	17-3
Description	17-3
Settings	17-3
Tip	17-3
Command-Line Information	17-3
Recommended Settings	17-4
Refresh package list	17-5
Description	17-5
Tip	17-5
Initialize/Terminate	17-6
Description	17-6
Settings	17-6
Command-Line Information	17-6
Recommended Settings	17-6
Execution	17-7
Description	17-7
Settings	17-7
Command-Line Information	17-7
Recommended Settings	17-7
Shared utility	17-9
Description	17-9
Settings	17-9
Command-Line Information	17-9
Recommended Settings	17-9
Constants	17-11
Description	17-11
Settings	17-11

Command-Line Information	17-11
Recommended Settings	17-11
Inputs/Outputs	17-13
Description	17-13
Settings	17-13
Command-Line Information	17-13
Recommended Settings	17-13
Internal data	17-15
Description	17-15
Settings	17-15
Command-Line Information	17-15
Recommended Settings	17-15
Parameters	17-17
Description	17-17
Settings	17-17
Command-Line Information	17-17
Recommended Settings	17-17
Validation results	17-19
Description	17-19
Settings	17-19
Recommended Settings	17-19

18 | Code Generation Parameters: Templates

Model Configuration Parameters: Code Generation Templates	18-2
Code Generation: Templates Tab Overview	18-3
Configuration	18-3
To get help on an option	18-3

19 | Code Generation Parameters: Verification

Model Configuration Parameters: Code Generation Verification	19-2
Code Generation: Verification Tab Overview	19-3
Configuration	19-3
To get help on an option	19-3

Recommended Settings Summary for Model Configuration Parameters

..... 20-2

Embedded Coder Model Advisor Checks

Embedded Coder Checks	21-2
Check for blocks not recommended for C/C++ production code deployment	21-3
Check configuration parameters for generation of inefficient saturation code	21-4
Identify lookup table blocks that generate expensive out-of-range checking code	21-5
Check output types of logic blocks	21-6
Check the hardware implementation	21-7
Identify questionable software environment specifications	21-8
Identify questionable code instrumentation (data I/O)	21-9
Identify blocks generating inefficient algorithms	21-10
Check for unsupported block names	21-11
Check usage of Assignment blocks	21-12
Check for switch case expressions without a default case	21-13
Check for missing error ports for AUTOSAR receiver interfaces	21-14
Check bus object names that are used as bus element names	21-15
Check configuration parameters for secure coding standards	21-16
Check for blocks not recommended for secure coding standards	21-18
Identify questionable subsystem settings	21-19
Check usage of 'long long' data type	21-20
Check for blocks not supported for row-major code generation	21-20
Identify TLC S-Functions with unset array layout	21-21
Identify blocks that generate expensive fixed-point and saturation code	21-22
Check for missing const qualifiers in model functions	21-24
Identify questionable fixed-point operations	21-24
Identify blocks that generate expensive rounding code	21-26
Check for bitwise operations on signed integers	21-27
Check for recursive function calls	21-28
Check for equality and inequality operations on floating-point values ..	21-28
Check integer word lengths	21-29
Check block names	21-30
Check modeling style for component deployment	21-31
Check signal interfaces	21-32
Check Startup and Shutdown Event	21-33
Check usage of partial data send	21-34
Check configuration for component deployment	21-34
Check for variant blocks that do not have a default choice	21-35

22	Simulink Coder Tools
23	Embedded Coder Tools
24	C/C++ Functions That Support Symbolic Dimensions for Simulink Function Blocks
25	Apps

Embedded Coder Functions

activateConfigSet

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, *Simulink.ConfigSet*, which already exists in the model. The original configuration set for the model is restored after execution of the *cgv.CGV* object.

See Also

Topics

“Activate a Configuration Set”

“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'

Version History

Introduced in R2007b

See Also

[addAdditionalIncludePath](#) | [addAdditionalSourceFile](#) | [addAdditionalSourcePath](#)

Topics

“Define Code Replacement Library Optimizations”

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;
% custom code defining entry
% .
% .
% .
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')`

Version History

Introduced in R2007b

See Also

`addAdditionalHeaderFile` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

Topics

“Define Code Replacement Library Optimizations”

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'`

Version History

Introduced in R2007b

See Also

`addAdditionalLinkObjPath`

Topics

“Define Code Replacement Library Optimizations”

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`

Version History

Introduced in R2008a

See Also

`addAdditionalLinkObj`

Topics

“Define Code Replacement Library Optimizations”

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'

Version History

Introduced in R2007b

See Also

[addAdditionalHeaderFile](#) | [addAdditionalIncludePath](#) | [addAdditionalSourcePath](#)

Topics

"Define Code Replacement Library Optimizations"

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')`

Version History

Introduced in R2007b

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalSourceFile`

Topics

"Define Code Replacement Library Optimizations"

addAlgorithmProperty

Add algorithm properties for code replacement table entry

Syntax

addAlgorithmProperty(hEntry,Name,Value)

Description

addAlgorithmProperty(hEntry,Name,Value) 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

Configure Algorithm Properties for lookup1D Function

This example shows how to use the addAlgorithmProperty function to configure the code generator to apply these 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');
```

Input Arguments

hEntry — Code replacement table entry

object

Code replacement table entry, previously returned by calling the function `RTW.TfLCFunctionEntry` or `RTW.TfLCOperationEntry`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `'Name', 'y1'`

ApplyFullPrecisionForLinearInterpolation — Specify whether or not to replace generated lookup table functions that use full-precision fixed-point algorithm

`'off'` (default) | `'on'`

In a lookup table block, the `ApplyFullPrecisionForLinearInterpolation` value specifies whether or not to use full-precision fixed-point algorithm for linear interpolation lookup when possible. For a code replacement entry, the value specifies whether to match generated lookup functions that use full-precision table lookup.

If you set `ApplyFullPrecisionForLinearInterpolation` to `on` in a code replacement entry:

- If the value on the block is set to `on` but the code generator was not able to use the full-precision fixed-point algorithm for the generated lookup function, the entry does not match the generated function.
- A match can cause a numerical mismatch between simulation and code generation if the implementation algorithm is different from the algorithm used for simulation.

BPPower2Spacing — Specify whether or not to replace blocks whose breakpoints are power of 2 spaced integer values

`'off'` (default) | `'on'`

Set the `BPPower2Spacing` value to `on` if you want to replace blocks whose breakpoints are the power of 2 spaced values by the user-defined function specified in the code replacement library. Set the value to `off` if you want to replace blocks whose breakpoints are not the power of 2 spaced values by the user-defined function specified in the code replacement library.

ExtrapMethod — Specify the method to use to extrapolate values

`'Linear'` (default) | `'Clip'`

The `ExtrapMethod` value specifies which method to use to extrapolate values for inputs that fall outside the range of a breakpoint data set.

Example: `'ExtrapMethod', 'Clip'`

IndexSearchMethod — Specify the method of calculating table indices

`'Binary search'` (default) | `'Linear search'` | `'Evenly spaced points'`

The `IndexSearchMethod` value specifies the method of calculating table indices. For evenly spaced breakpoint sets, you achieve optimal speed by selecting `Evenly spaced points` to calculate table indices. For unevenly spaced breakpoint sets, use `Linear search` or `Binary search`.

Example: `'IndexSearchMethod', 'Evenly spaced points'`

InterpMethod — Specify the method of interpolation between breakpoint values

`'Linear point-slope'` (default) | `'Linear Lagrange'` | `'Flat'` | `'Nearest'`

The `InterpMethod` value specifies the method of interpolation between breakpoint values.

Example: `'InterpMethod', 'Linear point-slope'`

RemoveProtectionInput — Specify whether or not to include code that checks for out-of-range input values

`'off'` (default) | `'on'`

The `RemoveProtectionInput` value specifies whether or not to include code that checks for out-of-range input values. If your input values are not out of range, you can set the `RemoveProtectionInput` to `off`.

Example: `'RemoveProtectionInput', 'off'`

RndMeth — Specify the rounding mode for fixed-point operations

`'Simplest'` (default) | `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Zero'`

The `RndMeth` value specifies the rounding mode for fixed-point lookup table calculations.

Example: `'RndMeth', 'Zero'`

SaturateOnIntegerOverflow — Specify whether to use data to saturate on integer overflow

`'off'` (default) | `'on'`

The `SaturateOnIntegerOverflow` value specifies the method of overflow action. If you set the value to `off`, overflows wrap to the appropriate value that is represented by the data type. Otherwise, overflows saturate to either the minimum or maximum value that the data type can represent.

Example: `'SaturateOnIntegerOverflow', 'on'`

SupportTunableTableSize — Specify whether or not to enable tunable table size in the generated code

`'off'` (default) | `'on'`

If you want to enable tunable table size in the generated code, set the `SupportTunableTableSize` value to `on`.

Example: `'SupportTunableTableSize', 'on'`

UseLastTableValue — Specify indexing convention

`'off'` (default) | `'on'`

The `UseLastTableValue` value specifies the indexing convention to use to address the last element of a breakpoint set and its corresponding table value.

Example: `'UseLastTableValue', 'on'`

UseRowMajorAlgorithm — Specify whether or not to use row-major algorithm

'off' (default) | 'on'

Set the UseRowMajorAlgorithm value to on to enable using row-major algorithms. Set the value to off to use column-major algorithms.

Example: 'UseRowMajorAlgorithm', 'on'

Version History

Introduced in R2014b

See Also

getTflArgFromString

Topics

“Lookup Table Function Code Replacement”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

addBaseline

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 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 *addInputData*.
- 3 Add the baseline file to the *cgv.CGV* object by calling *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 *run*, which automatically compares the baseline data to the result data in this execution.
- 5 Call *getStatus* to determine the results of the comparison.

See Also

[addInputData](#) | [run](#) | [createToleranceFile](#) | [getStatus](#)

Topics

[“Verify Numerical Equivalence with CGV”](#)

addHeaderReportFcn

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

run

Topics

“Customize Model Behavior with Callbacks”

addPostExecFcn

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

run

Topics

“Customize Model Behavior with Callbacks”

addPostExecReportFcn

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

run

Topics

“Customize Model Behavior with Callbacks”

addPreExecFcn

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

run

Topics

“Customize Model Behavior with Callbacks”

addPreExecReportFcn

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

run

Topics

“Customize Model Behavior with Callbacks”

addTrailerReportFcn

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

run

Topics

“Customize Model Behavior with Callbacks”

addCheck

Namespace: rtw.codegenObjectives

Add check to code generation objective

Syntax

addCheck(objective, checkID)

Description

addCheck(objective, checkID) adds the specified check to the specified objective in the Code Generation Advisor. When you select the objective, the Code Generation Advisor includes the check, unless another objective with a higher priority excludes the check.

Examples

Create a Custom Code Generation Objective

Create a custom objective named **Reduced RAM Example** that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

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

```
%Register the objective  
register(obj);  
  
end
```

Input Arguments

objective — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

checkID — Identifier of check

character vector | string scalar

Identifier of check that you want to add, specified as a character vector or string scalar.

Example: `'mathworks.codegen.CodeInstrumentation'`

Version History

Introduced in R2009a

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. This value specifies the alignment boundary.

Example: 16

Version History

Introduced in R2014a

See Also

Topics

"Data Alignment for Code Replacement"
"Define Code Replacement Library Optimizations"
"Code You Can Replace from MATLAB Code"
"Code You Can Replace From Simulink Models"

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')'`

Version History

Introduced in R2007b

See Also

`getTflArgFromString`

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

addDWorkArg

Add DWork argument for entry in code replacement table

Syntax

```
addDWorkArg(hEntry,arg)
```

Description

addDWorkArg(hEntry,arg) adds a specified DWork argument to the arguments for a semaphore or block 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, by using *hEntry* = RTW.TfLCSemaphoreEntry, or a block replacement entry class, by using *hEntry* = RTW.TfLBlockEntry.

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)` or `createDWorkArg`, added to the arguments for the code replacement table entry.

Example: `'hLib.getTfLDWorkFromString('dl','void*')`

Example: `'createDWorkArg(blockEntry,'Pointer','Name','crlDWorkArg')`

Version History

Introduced in R2011b

See Also

`getTfLDWorkFromString` | `createDWorkArg`

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

“Block Replacement for Code Optimization”

addConfigSet

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

“Manage Configuration Sets for a Model”

“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 code replacement 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, operator, or block entry

handle

The *entry* is a handle to a function, operator, or block entry that you have constructed by calling *hEntry* = RTW.TflCFunctionEntry, *hEntry* = RTW.TflCOperationEntry, or *hEntry* = RTW.TflBlockEntry.

Example: op_entry

Version History

Introduced in R2007b

See Also

RTW.TflTable

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

addInputData

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

run

Topics

“Verify Numerical Equivalence with CGV”

addParam

Namespace: rtw.codegenObjectives

Add parameters to code generation objective

Syntax

addParam(objective, param, value)

Description

addParam(objective, param, value) adds the specified 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**.

Examples

Create a Custom Code Generation Objective

Create a custom objective named Reduced RAM Example that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file sl_customization.m to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the addObjectives function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

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

```
%Register the objective  
register(obj);  
  
end
```

Input Arguments

objective — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

param — Name of parameter

character vector | string scalar

Name of parameter to add to the objective, specified as a character vector or string scalar.

value — Parameter value

character vector | string scalar

Parameter value to verify in the Code Generation Advisor, specified as a character vector or string scalar.

Version History

Introduced in R2009a

See Also

`get_param`

Topics

“Create Custom Code Generation Objectives”

addPostLoadFiles

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”

“Customize Model Behavior with Callbacks”

annotate

Namespace: `coder.profile`

Color profiled model components or open model with profiled components colored

Syntax

```
annotate(myExecutionProfile)
```

Description

`annotate(myExecutionProfile)` 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.

Examples

Annotate Model

To annotate the profiled model, use the `annotate` function and the `executionProfile` workspace variable.

```
annotate(executionProfile);
```

Input Arguments

myExecutionProfile — Specifies annotation

workspace variable

When you run a SIL, PIL, or XCP-based external mode simulation with code execution profiling, the software generates the workspace variable `executionProfile`, which you specify in **Configuration Parameters > Code Generation > Verification > Workspace variable**.

Example: `executionProfile`

Version History

Introduced in R2016b

See Also

`report`

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

cgv.CGV

Represent a model as a code generation verification object that has methods to configure and execute a model in a variety of simulation modes to automate verification of numerical equivalence of executing the generated code

Description

A code generation verification (cgv) object 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.

Creation

`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.

The `cgv.CGV` function accepts these 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

If `topmodel` (default), top-model SIL or PIL simulation mode and standalone code interface.

If `modelblock`, model block SIL or PIL simulation mode and model reference code interface mode.

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

- `Connectivity` — Specify mode of execution

If `sim` or `normal` (default), mode of execution is Normal simulation.

If `sil`, mode of execution is SIL.

If `pil`, mode of execution is PIL.

Example: “Test Model for Numerical Equivalence” on page 1-44

Properties

Description — Object description

' ' (null character vector) (default)

Specify a description of the object.

Name — Object name

' ' (null character vector) (default)

Specify a name for the object.

Object Functions

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

Examples

Test Model for Numerical Equivalence

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

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`

Run the model for each mode of execution using the `cgvObj.run` method.

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.

Version History

Introduced in R2009b

See Also

`cgv.Config`

Topics

“Verify Numerical Equivalence with CGV”

cgv.Config

Check and modify model configuration parameter values

Description

A code generation verification configuration (`cgv.Config`) 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 “Customize Model Behavior with Callbacks”.

Creation

`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.

The `cgv.Config` function accepts these 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 an argument name or an 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

CheckOutputs — Check for compatible outputs

on (default) | off

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

- If on, compile the model and check the model outputs configuration.
- If off, do not compile the model or check the model outputs configuration.

Example: 'CheckOutputs', 'on'

ComponentType — Define the SIL or PIL approach

topmodel (default) | modelblock

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`.

If `topmodel`, the top-model SIL or PIL simulation mode and standalone code interface.

If `modelblock`, model block SIL or PIL simulation mode and model reference code interface.

Example: 'ComponentType', 'topmodel'

Connectivity — Specify mode of execution

sim (default) | sil | pil

If `sim`, the mode of execution is simulation. Recommends changes to a subset of the configuration parameters that SIL and PIL targets require.

If `sil`, the 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.

If `pil`, the 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.

Example: 'Connectivity', 'sim'

LogMode — Select log mode

SignalLogging | SaveOutput

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

If set to `SignalLogging`, the object logs 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.

If set to `SaveOutput`, the object saves 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.

Example: 'SignalLogging', ''

ReportOnly — Select report or modify configuration

off (default) | on

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.

Example: 'ReportOnly', 'off'

SaveModel — Select save model

off (default) | on

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'.

Example: 'SaveModel', 'off'

Object Functions

<code>configModel</code>	Determine and change configuration parameter values
<code>displayReport</code>	Display results of comparing configuration parameter values
<code>getReportData</code>	Return results of comparing configuration parameter values

Examples

Configure Model for Top-Model SIL

Configure the RollAxisAutopilot 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.
openExample('ecoder/TraceGeneratedCodeToBlocksExample','supportingFile','RollAxisAutopilot')
set_param('RollAxisAutopilot', 'SaveFormat', 'StructureWithTime');
cgvCfg = cgv.Config('RollAxisAutopilot', 'LogMode', 'SaveOutput', ...
    'connectivity', 'sil');
cgvCfg.configModel();

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

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

Version History

Introduced in R2009b

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`).

Note To use data alignment, you must specify the alignment information for the compiler by using a code replacement library registration. For more information, see “Provide Data Alignment Specifications for Compilers”.

Examples

Data Alignment for Exported Data

Generate code for an example function that specifies data alignment for exported data.

Create a function `exportedDataExampleFun.m` that aligns the data for a global variable `z` along an 8-byte boundary.

```
function a = exportedDataExampleFun(b)

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

a = b + z;

end
```

Create a wrapper function `wrapperFun.m` that calls the example function.

```
function out1 = wrapperFun()

a = double([4,5;6,7]);

out1 = exportedDataExampleFun(a);

end
```

Create the following code replacement library file named `crtl_table_align.m`. You use this table because data alignment compiler specification requires that you register a code replacement library. The table does not contain code replacement entries, it only specifies the compiler information needed for data alignment.

```
function hTable = crt_table_align
```

```
%% Create a table object  
hTable = RTW.TflTable;
```

Create and save the following registration file, `rtwTargetInfo.m`. The file specifies the data alignment information for the compiler and associates that information with the code replacement library that you created.

```
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 crt_table_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);
```

```
% 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 = {'crt_table_align'};  
thisCrl.TargetCharacteristics = tc;
```

```
end % End of LOCCRLREGFCN
```

Register the code replacement library.

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

Create a code generation configuration that uses the code replacement library that you created and registered with the data alignment specification.

```
config = coder.EmbeddedCodeConfig;  
config.CodeReplacementLibrary = "Data Alignment Example";
```

Generate code for the function using the global variable `z`.

```
global z;
z = double([1,2;3,4]);
codegen -config config wrapperFun.m
```

Examine the generated code. For this example, the global variable `z` is defined in the file `wrapperFun_data.c` with the alignment specification.

```
/* Variable Definitions */
__attribute__((aligned(8))) double z[4];
```

For comparison, when you generate code for this function without using data alignment, the global variable is defined without the alignment specification.

```
/* Variable Definitions */
double z[4];
```

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. 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.
- The `coder.dataAlignment` function generates an error if a code replacement library is not specified. For more information, see “Provide Data Alignment Specifications for Compilers”.

For more information about general limitations on data alignment, see “Data Alignment Limitations”.

Tips

- To prevent inlining a function containing an aligned variable in the generated code, a best practice is to call `coder.inline('never')` in your code.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`codegen`

Topics

[“Data Alignment for Code Replacement”](#)

[“Define Code Replacement Library Optimizations”](#)

[“What Is Code Replacement Customization?”](#)

[“What Is Code Replacement?”](#)

coder.descriptor.DataTransferElement Class

Namespace: coder.descriptor

Return information about data transfer signal

Description

An object of the `coder.descriptor.DataTransferElement` class represents the data transfer signal that connects communicating functions. You can configure the mapping of a data transfer signal on the **Data Transfers** tab in the Code Mappings editor or the code mappings programming interface.

Properties

Name — Signal name

character vector | string scalar

Data transfer signal name, specified as a character vector or string scalar.

Attributes:

GetAccess	public
SetAccess	public

InitialCondition — Initial value

character vector | string scalar

Initial value of the signal, specified as a character vector or string scalar.

Attributes:

GetAccess	public
SetAccess	public

DataCommunicationMethod — Data communication method

OutsideExecution | DuringExecution | DirectAccess

Data communication method for the data transfer signal, specified as `OutsideExecution`, `DuringExecution`, or `DirectAccess`. You can set this value in the Embedded Coder Dictionary when you define the interface.

Attributes:

GetAccess	public
SetAccess	public

Functions — Data transfer functions

array of `coder.descriptor.DataTransferFunction` objects

Data transfer functions that the data transfer signal interacts with, specified as an array of `coder.descriptor.DataTransferFunction` objects.

Attributes:

GetAccess

public

Version History

Introduced in R2022b

See Also

`coder.descriptor.DataTransferFunction` |
`coder.descriptor.DataTransferServiceInterface`

Topics

“Get Metadata About Service Interface”
“Data Transfer Service Interfaces”

coder.descriptor.DataTransferFunction Class

Namespace: coder.descriptor

Superclasses: coder.descriptor.FunctionInterface

Return information about function prototypes for data transfer service interfaces

Description

An object of the `coder.descriptor.DataTransferFunction` class represents the data transfer function interface in the generated code.

Properties

Prototype — Description of function prototype

`coder.descriptor.Prototype` object

Description of the function prototype, including the function return value, name, arguments, header, and source file, specified as a `coder.descriptor.Prototype` object.

Attributes:

GetAccess	public
SetAccess	public

ActualReturn — Return arguments from function

`coder.descriptor.DataInterface` object

Return arguments from the function, specified as a `coder.descriptor.DataInterface` object. When the function returns no data, this field is empty.

Attributes:

GetAccess	public
SetAccess	public

VariantInfo — Variant conditions in model

`coder.descriptor.VariantInfo` object

Variant conditions in the model that interact with the function interface, specified as a `coder.descriptor.VariantInfo` object.

Attributes:

GetAccess	public
SetAccess	public

FunctionOwner — Function owner

`coder.descriptor.TypedRegion` object

Function owner of the class that the function belongs to, specified as a `coder.descriptor.TypedRegion` object. For C code generation, this property is empty.

Attributes:

GetAccess	public
SetAccess	public

Timing — Function access rate in target environment`coder.descriptor.TimingInterface` object

Function access rate in the target environment, specified as a `coder.descriptor.TimingInterface` object.

Attributes:

GetAccess	public
SetAccess	public

ActualArgs — Input arguments to the function`coder.descriptor.DataInterfaceList` object

Input arguments to the function, specified as a `coder.descriptor.DataInterfaceList` object. When there is no data passed as an argument to the function, this property is empty.

Attributes:

GetAccess	public

FunctionType — Function type`Set | Get`

Type of access function, specified as an enumerated value of `Set` or `Get`.

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2022b**See Also**

`coder.descriptor.DataTransferElement` |
`coder.descriptor.DataTransferServiceInterface`

Topics

“Get Metadata About Service Interface”
“Data Transfer Service Interfaces”

coder.descriptor.DataTransferServiceInterface Class

Namespace: `coder.descriptor`

Return information about data transfer service interface

Description

An object of the `coder.descriptor.DataTransferServiceInterface` class represents the data transfer service interface configuration.

Creation

`serviceInterfaceTypeObj = getServiceInterface(serviceObj, serviceInterfaceType)` creates a data transfer service interface configuration for the service interface `serviceObj` of the type `serviceInterfaceType`.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceInterfaceType — Service interface type

`coder.descriptor.Services.DataTransfer`

Data transfer service interface type specified in the Embedded Coder Dictionary, specified as an enumerated value of `coder.descriptor.Services.DataTransfer`.

Properties

DataTransferElements — Data transfer signal

array of `coder.descriptor.DataTransferElement` objects

Data transfer signal that connects the communicating functions, specified as an array of `coder.descriptor.DataTransferElement` objects.

Attributes:

GetAccess public

Version History

Introduced in R2022b

See Also

`coder.descriptor.DataTransferElement` | `coder.descriptor.DataTransferFunction`

Topics

“Get Metadata About Service Interface”

“Data Transfer Service Interfaces”

coder.descriptor.MeasurementServiceInterface Class

Namespace: `coder.descriptor`

Return information about measurement service interfaces

Description

An object of the `coder.descriptor.MeasurementServiceInterface` class represents the measurement service configurations defined in the Embedded Coder Dictionary.

Creation

`serviceInterfaceTypeObj = getServiceInterface(serviceObj, serviceInterfaceType)` creates a `coder.descriptor.MeasurementServiceInterface` object that represents the measurement service interface configurations.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceInterfaceType — Sender and receiver service interface type

enumerated value of `coder.descriptor.Services.Measurement`

Sender and receiver service interface type defined in the Embedded Coder Dictionary, specified as an enumerated value of `coder.descriptor.Services.Measurement`.

Methods

Public Methods

<code>getDataStoreInterfaces</code>	Return information about datastore measurement service interfaces
<code>getPersistentDataInterfaces</code>	Return information about datastore and state measurement service interfaces with persistent data specification
<code>getSignalInterfaces</code>	Return information about signal measurement service interfaces
<code>getStateInterfaces</code>	Return information about state measurement service interfaces

Version History

Introduced in R2023a

See Also

`coder.descriptor.ServiceInterface` | `getServiceInterface`

Topics

“Get Metadata About Service Interface”

“Generate C Measurement Service Interface Code for Component Deployment”

coder.descriptor.ParameterTuningServiceInterface Class

Namespace: `coder.descriptor`

Return information about parameter tuning service interfaces

Description

An object of the `coder.descriptor.ParameterTuningServiceInterface` class represents the parameter tuning service interface configurations defined in the Embedded Coder Dictionary.

Creation

`serviceInterfaceTypeObj = getServiceInterface(serviceObj, serviceInterfaceType)` creates a `coder.descriptor.ParameterTuningServiceInterface` object that represents the parameter tuning service interface configurations.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceInterfaceType — Parameter tuning service interface type

enumerated value of `coder.descriptor.Services.ParameterTuning`

Parameter tuning service interface type defined in the Embedded Coder Dictionary, specified as an enumerated value of `coder.descriptor.Services.ParameterTuning`.

Methods

Public Methods

<code>getParameterInterfaces</code>	Return information about parameter tuning service interfaces
<code>getParameterArgumentInterfaces</code>	Return information about parameter argument tuning service interfaces

Version History

Introduced in R2023a

See Also

`coder.descriptor.ServiceInterface` | `getServiceInterface`

Topics

"Get Metadata About Service Interface"

"Generate C Parameter Tuning Service Interface Code for Component Deployment"

"Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments"

coder.descriptor.SenderReceiverServiceInterface Class

Namespace: `coder.descriptor`

Return information about sender and receiver service interfaces

Description

An object of the `coder.descriptor.SenderReceiverServiceInterface` class represents the sender and receiver service interface configurations defined in the Embedded Coder Dictionary.

Creation

`serviceInterfaceTypeObj = getServiceInterface(serviceObj, serviceInterfaceType)` creates a `coder.descriptor.SenderReceiverServiceInterface` object that represents the sender and receiver service interface configurations.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceInterfaceType — Service interface type

enumerated value of `coder.descriptor.Services.SenderReceiver`

Sender and receiver service interface type specified in the Embedded Coder Dictionary, specified as an enumerated value of `coder.descriptor.Services.SenderReceiver`.

Methods

Public Methods

<code>getReceiverInterfaces</code>	Return information about receiver service interfaces
<code>getSenderInterfaces</code>	Return information about sender service interfaces

Version History

Introduced in R2022b

See Also

`coder.descriptor.ServiceInterface` | `getServiceInterface`

Topics

“Get Metadata About Service Interface”

coder.descriptor.ServiceFunctions Class

Namespace: `coder.descriptor`

Return list of service functions that entry-point function calls

Description

An object of the `coder.descriptor.ServiceFunctions` class represents the list of service functions that a generated entry-point function calls.

Creation

`serviceFunctions = getCalledServiceFunctions(serviceObj, entryPointFunctionName)` creates a `ServiceFunction` object that returns a list of service functions that the entry-point function, `entryPointFunctionName`, calls. For more information, see `getCalledServiceFunctions`.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated entry-point function, specified as a `coder.descriptor.ServiceInterface` object.

entryPointFunctionName — Entry-point function

character vector | string scalar

Entry-point function for which you want to retrieve service function calls, specified as a character vector or string scalar.

Properties

ReceiverFunctions — Receiver service function

set of string

Receiver service function that is produced in the generated code, specified as a string.

Attributes:

GetAccess public

SenderFunctions — Sender service function

set of string

Sender service function that is produced in the generated code, specified as a string.

Attributes:

GetAccess public

DataTransferFunctions — Data transfer service function

set of string

Data transfer service function that is produced in the generated code, specified as a string.

Attributes:

GetAccess public

TimerFunctions — Timer service function

set of string

Timer service function that is produced in the generated code, specified as a string.

Attributes:

GetAccess public

Version History

Introduced in R2022b

See Also

coder.descriptor.ServiceInterface | coder.descriptor.DataTransferFunction |
coder.descriptor.TimerFunction

Topics

“Get Metadata About Service Interface”
“Service Interfaces”

coder.descriptor.ServiceInterface Class

Namespace: `coder.descriptor`

Return service interface information required to generate service functions

Description

An object of the `coder.descriptor.ServiceInterface` class represents the service interface configuration that you define in the Embedded Coder Dictionary. Use `ServiceInterface` objects to retrieve metadata about a specified service function interface.

Creation

`serviceObj = getServices(codeDescObj)` creates a `ServiceInterface` object, from the `coder.codedescriptor.CodeDescriptor` object, `codeDescObj`.

Input Arguments

codeDescObj — Code descriptor

`coder.codedescriptor.CodeDescriptor` object

Code descriptor for which you want to retrieve the information about generated code, specified as a `coder.codedescriptor.CodeDescriptor` object.

Methods

Public Methods

<code>getCallableFunctionsThatCallServiceFunction</code>	Return entry-point functions that call service function
<code>getCalledServiceFunctions</code>	Returns service functions called from entry-point function
<code>getServiceDataCommMethod</code>	Return data communication method for service function
<code>getServiceInterface</code>	Return service interface object for service interface type
<code>getServicesHeaderFileName</code>	Return name of header file that contains service function declarations

Version History

Introduced in R2022b

See Also

`coder.codedescriptor.CodeDescriptor` |
`coder.descriptor.DataTransferServiceInterface` |
`coder.descriptor.MeasurementServiceInterface` |
`coder.descriptor.ParameterTuningServiceInterface` |

coder.descriptor.SenderReceiverServiceInterface |
coder.descriptor.TimerServiceInterface

Topics

“Get Metadata About Service Interface”

“Create a Service Interface Configuration”

coder.descriptor.TimerFunction Class

Namespace: `coder.descriptor`

Retrieve information about function prototypes for timer services

Description

An object of the `coder.descriptor.TimerFunction` class represents a timer function in the generated code.

Properties

Prototype — Description of function prototype

`coder.descriptor.Prototype` object

Description of the function prototype, including the function return value, name, arguments, header, and source file, specified as a `coder.descriptor.Prototype` object.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

DataCommunicationMethod — Data communication method

`DirectAccess` | `DuringExecution` | `OutsideExecution`

Data communication method for the service function as configured in Embedded Coder Dictionary, specified as `DirectAccess` , `DuringExecution`, or `OutsideExecution`.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Timing — Function access rate in target environment

`coder.descriptor.TimingInterface` object

Function access rate in the target environment, specified as a `coder.descriptor.TimingInterface` object.

Attributes:

<code>GetAccess</code>	<code>public</code>

Version History

Introduced in R2022b

See Also

`coder.descriptor.TimerServiceInterface` | `getServiceDataCommMethod`

Topics

"Get Metadata About Service Interface"

"Timer Service Interfaces"

coder.descriptor.TimerServiceInterface Class

Namespace: `coder.descriptor`

Return information about timer service interface

Description

An object of the `coder.descriptor.TimerServiceInterface` class represents the timer service interface configuration.

Creation

`serviceInterfaceTypeObj = getServiceInterface(serviceObj, serviceInterfaceType)` represents timer service interface configuration for the service interface `serviceObj` of type `serviceInterfaceType`.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceInterfaceType — Service interface type

`coder.descriptor.Services.Timer`

Timer service interface type defined in Embedded Coder Dictionary, specified as an enumerated value of `coder.descriptor.Services.Timer`.

Properties

TimerFunctions — Function prototype of timer services

array of `coder.descriptor.TimerFunction` objects

Function prototype for calling timer service, specified as an array of `coder.descriptor.TimerFunction` objects.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Version History

Introduced in R2022b

See Also

`getSenderInterfaces` | `coder.descriptor.TimerFunction`

Topics

“Get Metadata About Service Interface”

“Timer Service Interfaces”

getServiceFunctionDeclaration

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `coder.codedescriptor`

Return declaration of service function interface in generated code

Syntax

```
serviceFunctionDeclaration = getServiceFunctionDeclaration(codeDescObj,  
serviceFunctionPrototype)
```

Description

`serviceFunctionDeclaration = getServiceFunctionDeclaration(codeDescObj, serviceFunctionPrototype)` returns the function declaration of the service interface prototype `serviceFunctionPrototype` as produced by the generated code in the code descriptor object `codeDescObj`.

Input Arguments

codeDescObj — Code descriptor

`coder.codedescriptor.CodeDescriptor` object

Code descriptor for which you want to retrieve the information about generated code, specified as a `coder.codedescriptor.CodeDescriptor` object.

serviceFunctionPrototype — Service function prototype

`coder.descriptor.types.Prototype` object

Service function prototype, specified as `coder.descriptor.types.Prototype` object. Use the `getServiceFunctionPrototype` method to retrieve the `Prototype` object for the service function.

Output Arguments

serviceFunctionDeclaration — Service function declaration

character vector | string scalar

Service function declaration produced in the generated code, returned as a character vector or string scalar.

Version History

Introduced in R2022b

See Also

`getServiceFunctionPrototype` | `getServices`

Topics

“Get Metadata About Service Interface”

getServiceFunctionPrototype

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `coder.codedescriptor`

Return prototype of generated service function interface

Syntax

```
serviceFunctionPrototype = getServiceFunctionPrototype(codeDescObj,  
serviceFunctionName)
```

Description

`serviceFunctionPrototype = getServiceFunctionPrototype(codeDescObj, serviceFunctionName)` returns the prototype of the service function `serviceFunctionName` for the code descriptor `codeDescObj`. The returned `coder.descriptor.types.Prototype` object contains properties such as the name of the service function, information about arguments, and header and source file.

Input Arguments

codeDescObj — Code descriptor

`coder.codedescriptor.CodeDescriptor` object

Code descriptor for which you want to retrieve the information about generated code, specified as a `coder.codedescriptor.CodeDescriptor` object.

serviceFunctionName — Service function name

character vector | string scalar

Service function name, specified as character vector or string scalar.

Output Arguments

serviceFunctionPrototype — Service function prototype

`coder.descriptor.types.Prototype` object

Service function prototype for the service function, returned as a `coder.descriptor.types.Prototype` object. This object is empty if the `serviceFunctionName` input argument is invalid or does not exist in the generated code.

Version History

Introduced in R2022b

See Also

`getServiceFunctionDeclaration` | `getServices`

Topics

“Get Metadata About Service Interface”

getServices

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `coder.codedescriptor`

Return service interface object

Syntax

```
serviceObj = getServices(codeDescObj)
```

Description

`serviceObj = getServices(codeDescObj)` returns the service interface specified as a `coder.descriptor.ServiceInterface` object. Use the `coder.descriptor.ServiceInterface` object to call associated methods that retrieve metadata about a specified service function.

Input Arguments

codeDescObj — Code descriptor

`coder.codedescriptor.CodeDescriptor` object

Code descriptor for which you want to retrieve the information about generated code, specified as a `coder.codedescriptor.CodeDescriptor` object.

Output Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

Version History

Introduced in R2022b

See Also

`getServiceFunctionPrototype` | `getServiceFunctionDeclaration`

Topics

“Get Metadata About Service Interface”

getStoreInterfaces

Class: `coder.descriptor.MeasurementServiceInterface`

Namespace: `coder.descriptor`

Return information about datastore measurement service interfaces

Syntax

```
dataStoreInterfaces = getStoreInterfaces(measurementInterfaceObj)
```

Description

`dataStoreInterfaces = getStoreInterfaces(measurementInterfaceObj)` returns the `coder.descriptor.DataInterface` object that represents the datastore measurement service interface configuration.

Input Arguments

measurementInterfaceObj — Measurement service interface object

`coder.descriptor.MeasurementServiceInterface` object

Measurement service interface object, specified as a `coder.descriptor.MeasurementServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

dataStoreInterfaces — Datastore measurement service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

Datastore measurement service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object contains information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Version History

Introduced in R2023a

See Also

`coder.descriptor.SenderReceiverServiceInterface` | `getSenderInterfaces`

Topics

“Get Metadata About Service Interface”

“Generate C Measurement Service Interface Code for Component Deployment”

getPersistentDataInterfaces

Class: `coder.descriptor.MeasurementServiceInterface`

Namespace: `coder.descriptor`

Return information about datastore and state measurement service interfaces with persistent data specification

Syntax

```
persistentDataInterfaces = getPersistentDataInterfaces(  
measurementInterfaceObj)
```

Description

`persistentDataInterfaces = getPersistentDataInterfaces(measurementInterfaceObj)` returns an array of `coder.descriptor.DataInterface` objects representing all of the persistent data in the measurement service interface.

Input Arguments

measurementInterfaceObj — Measurement service interface object

`coder.descriptor.MeasurementServiceInterface` object

Measurement service interface object, specified as a `coder.descriptor.MeasurementServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

persistentDataInterfaces — Persistent data measurement service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

Persistent data measurement service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object contains information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Examples

Get Information About Persistent Data in Measurement Service Interface

Open the example and example model.

```
openExample('ecoder/GenerateMeasureServiceInterfaceCodeForPersistentDataExample')  
open_system('ComponentServiceWithPersistency');
```


The attached Embedded Coder Dictionary `ComponentServiceCoderDictionary.sldd` defines a measurement service interface with persistent data support enabled.

The Integrator subsystem contains a Discrete-Time Integrator block whose input is a signal of length 10. Configure the block so it has a different initial condition for each signal element.

```
set_param('ComponentServiceWithPersistency/Integrator/Discrete-Time Integrator', ...
    InitialCondition='1:10')
```

Build the model.

```
slbuild('ComponentServiceWithPersistency')
```

Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('ComponentServiceWithPersistency')
```

Create a `coder.codedescriptor.ServiceInterface` object from the code descriptor object.

```
serviceObj = getServices(codeDescObj);
```

Create a `coder.descriptor.MeasurementServiceInterface` object from the service interface object.

```
measurementServiceInterface = getServiceInterface(serviceObj, 'Measurement');
```

Retrieve persistent data interfaces from the measurement service interface object.

```
persistentDataInterfaces = getPersistentDataInterfaces(measurementServiceInterface);
```

```
persistentDataInterfaces =  
    1x2 ReadWriteDataInterface array with properties:
```

```
    DataReads  
    DataWrites  
    Type  
    SID  
    GraphicalName  
    VariantInfo  
    Implementation  
    Timing  
    Unit  
    Range
```

The method returns persistent data interface objects for the Unit Delay and Discrete-Time Integrator blocks. These blocks contain persistent data, such as initial conditions, that are stored in nonvolatile memory for subsequent time steps.

Get the initial condition of the Discrete-Time Integrator block.

```
get_param(persistentDataInterfaces(2).SID, 'InitialCondition')
```

```
ans =  
    '1:10'
```

Version History

Introduced in R2025a

See Also

`coder.descriptor.ServiceInterface | getServiceInterface`

Topics

“Get Metadata About Service Interface”

“Generate C Interface Code for Measurement Service Using Persistent Data for Component Deployment”

getSignalInterfaces

Class: `coder.descriptor.MeasurementServiceInterface`

Namespace: `coder.descriptor`

Return information about signal measurement service interfaces

Syntax

```
signalInterfaces = getSignalInterfaces(measurementInterfaceObj)
```

Description

`signalInterfaces = getSignalInterfaces(measurementInterfaceObj)` returns the `coder.descriptor.DataInterface` object that represents the signal measurement service interface configuration.

Input Arguments

measurementInterfaceObj — Measurement service interface object

`coder.descriptor.MeasurementServiceInterface` object

Measurement service interface object, specified as a `coder.descriptor.MeasurementServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

signalInterfaces — Signal measurement service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

Signal measurement service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object contains information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Version History

Introduced in R2023a

See Also

`coder.descriptor.SenderReceiverServiceInterface` | `getSenderInterfaces`

Topics

“Get Metadata About Service Interface”

“Generate C Measurement Service Interface Code for Component Deployment”

getStateInterfaces

Class: `coder.descriptor.MeasurementServiceInterface`

Namespace: `coder.descriptor`

Return information about state measurement service interfaces

Syntax

```
stateInterfaces = getStateInterfaces(measurementInterfaceObj)
```

Description

`stateInterfaces = getStateInterfaces(measurementInterfaceObj)` returns the `coder.descriptor.DataInterface` object that represents the state measurement service interface configuration.

Input Arguments

measurementInterfaceObj — Measurement service interface object

`coder.descriptor.MeasurementServiceInterface` object

Measurement service interface object, specified as a `coder.descriptor.MeasurementServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

stateInterfaces — State measurement service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

State measurement service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object contains information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Version History

Introduced in R2023a

See Also

`coder.descriptor.SenderReceiverServiceInterface` | `getSenderInterfaces`

Topics

“Get Metadata About Service Interface”

“Generate C Measurement Service Interface Code for Component Deployment”

getParameterArgumentInterfaces

Class: `coder.descriptor.ParameterTuningServiceInterface`

Namespace: `coder.descriptor`

Return information about parameter argument tuning service interfaces

Syntax

```
parameterArgumentTuningServiceInterfaces = getParameterArgumentInterfaces(  
parameterTuningInterfaceObj)
```

Description

`parameterArgumentTuningServiceInterfaces = getParameterArgumentInterfaces(
parameterTuningInterfaceObj)` returns the `coder.descriptor.DataInterface` object that represents the parameter argument tuning service interface configuration.

Input Arguments

parameterTuningInterfaceObj — Parameter tuning service interface object

`coder.descriptor.ParameterTuningServiceInterface` object

Parameter tuning service interface object, specified as a `coder.descriptor.ParameterTuningServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

parameterArgumentTuningServiceInterfaces — Parameter argument tuning service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

Parameter argument tuning service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object describes information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Version History

Introduced in R2023a

See Also

`coder.descriptor.SenderReceiverServiceInterface` | `getSenderInterfaces`

Topics

“Get Metadata About Service Interface”

“Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments”

getParameterInterfaces

Class: `coder.descriptor.ParameterTuningServiceInterface`

Namespace: `coder.descriptor`

Return information about parameter tuning service interfaces

Syntax

```
parameterTuningInterfaces = getParameterInterfaces(
parameterTuningInterfaceObj)
```

Description

`parameterTuningInterfaces = getParameterInterfaces(parameterTuningInterfaceObj)` returns the `coder.descriptor.DataInterface` object that represents the parameter tuning service interface configuration.

Input Arguments

parameterTuningInterfaceObj — Parameter tuning service interface object

`coder.descriptor.ParameterTuningServiceInterface` object

Parameter tuning service interface object, specified as a `coder.descriptor.ParameterTuningServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

parameterTuningInterfaces — Parameter tuning service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

Parameter tuning service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object describes information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Version History

Introduced in R2023a

See Also

`coder.descriptor.SenderReceiverServiceInterface` | `getSenderInterfaces`

Topics

“Get Metadata About Service Interface”

“Generate C Parameter Tuning Service Interface Code for Component Deployment”

“Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments”

getReceiverInterfaces

Class: `coder.descriptor.SenderReceiverServiceInterface`

Namespace: `coder.descriptor`

Return information about receiver service interfaces

Syntax

```
receiverInterfaces = getReceiverInterfaces(senderReceiverInterfaceObj)
```

Description

`receiverInterfaces = getReceiverInterfaces(senderReceiverInterfaceObj)` returns the `coder.descriptor.DataInterface` object that represents the receiver service interface configuration.

Input Arguments

senderReceiverInterfaceObj — Sender receiver service interface object

`coder.descriptor.SenderReceiverServiceInterface` object

Sender and receiver service interface object, specified as a `coder.descriptor.SenderReceiverServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

receiverInterfaces — Receiver service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

Receiver service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object describes information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Version History

Introduced in R2022b

See Also

`coder.descriptor.SenderReceiverServiceInterface` | `getSenderInterfaces`

Topics

“Get Metadata About Service Interface”

getSenderInterfaces

Class: `coder.descriptor.SenderReceiverServiceInterface`

Namespace: `coder.descriptor`

Return information about sender service interfaces

Syntax

```
senderInterfaces = getSenderInterfaces(senderReceiverInterfaceObj)
```

Description

`senderInterfaces = getSenderInterfaces(senderReceiverInterfaceObj)` returns the `coder.descriptor.DataInterface` object that represents the sender service interface configuration.

Input Arguments

senderReceiverInterfaceObj — Sender receiver service interface object

`coder.descriptor.SenderReceiverServiceInterface` object

Sender and receiver service interface object, specified as a `coder.descriptor.SenderReceiverServiceInterface` object. You can create this object by using the `getServiceInterface` method.

Output Arguments

senderInterfaces — Sender service interface

`coder.descriptor.DataInterface` object | array of `coder.descriptor.DataInterface` objects

Sender service interface, returned as a `coder.descriptor.DataInterface` object or an array of `coder.descriptor.DataInterface` objects. The `coder.descriptor.DataInterface` object describes information about the specified data interface, such as the type of data, Simulink identifier, graphical name, timing, implementation, and variant information.

Version History

Introduced in R2022b

See Also

`coder.descriptor.SenderReceiverServiceInterface` | `getReceiverInterfaces`

Topics

“Get Metadata About Service Interface”

getCallableFunctionsThatCallServiceFunction

Class: `coder.descriptor.ServiceInterface`

Namespace: `coder.descriptor`

Return entry-point functions that call service function

Syntax

```
entryPointFunctions = getCallableFunctionsThatCallServiceFunction(serviceObj,
serviceFunctionName)
```

Description

`entryPointFunctions = getCallableFunctionsThatCallServiceFunction(serviceObj, serviceFunctionName)` returns the names of the entry-point functions that call the specified service function.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceFunctionName — Service function name

character vector | string scalar

Service function name, specified as character vector or string scalar.

Output Arguments

entryPointFunctions — List of entry-point functions

character vector

List of entry-point functions that call the specified service function, returned as a character vector.

Version History

Introduced in R2022b

See Also

`coder.descriptor.ServiceInterface` | `getCalledServiceFunctions` | `getServiceDataCommMethod`

Topics

“Get Metadata About Service Interface”

getCalledServiceFunctions

Class: `coder.descriptor.ServiceInterface`

Namespace: `coder.descriptor`

Returns service functions called from entry-point function

Syntax

```
serviceFunctions = getCalledServiceFunctions(serviceObj,  
entryPointFunctionName)
```

Description

`serviceFunctions = getCalledServiceFunctions(serviceObj, entryPointFunctionName)` returns a list of service functions that the specified entry-point function, `entryPointFunctionName`, calls. The method returns the results as an array of strings that represent the `coder.descriptor.ServiceFunctions` object.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

entryPointFunctionName — Entry-point function

character vector | string scalar

Entry-point function for which you want to retrieve service function calls, specified as a character vector or string scalar.

Output Arguments

serviceFunctions — List of called service functions

`coder.descriptor.ServiceFunctions` object

List of called service functions from the specified entry-point function, `entryPointFunctionName`, returned as a `coder.descriptor.ServiceFunctions` object that contains properties that represent these types of service functions:

- `ReceiverFunctions`
- `SenderFunctions`
- `DataTransferFunctions`
- `TimerFunctions`

Version History

Introduced in R2022b

See Also

`coder.descriptor.ServiceInterface` | `getCallableFunctionsThatCallServiceFunction`
| `getServiceDataCommMethod`

Topics

"Get Metadata About Service Interface"

getServiceDataCommMethod

Class: coder.descriptor.ServiceInterface

Namespace: coder.descriptor

Return data communication method for service function

Syntax

```
dataCommMethod = getServiceDataCommMethod(serviceObj,serviceFunctionName)
```

Description

`dataCommMethod = getServiceDataCommMethod(serviceObj,serviceFunctionName)` returns the data communication method for the service function as configured in the Embedded Coder Dictionary. The result is an enumerated value of type `DataCommunicationMethodEnum`.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceFunctionName — Service function name

character vector | string scalar

Service function name, specified as character vector or string scalar.

Output Arguments

dataCommMethod — Data communication method

`DirectAccess` | `DuringExecution` | `OutsideExecution`

Data communication method for the service function as configured in Embedded Coder Dictionary, returned as an enumerated value of type `DataCommunicationMethodEnum`.

Version History

Introduced in R2022b

See Also

`coder.descriptor.ServiceInterface` | `getCallableFunctionsThatCallServiceFunction` | `getCalledServiceFunctions`

Topics

“Get Metadata About Service Interface”

“Create a Service Interface Configuration”

getServiceInterface

Class: `coder.descriptor.ServiceInterface`

Namespace: `coder.descriptor`

Return service interface object for service interface type

Syntax

```
serviceInterfaceTypeObj = getServiceInterface(serviceObj,  
serviceInterfaceType)
```

Description

`serviceInterfaceTypeObj = getServiceInterface(serviceObj, serviceInterfaceType)` returns the type of `coder.descriptor.ServiceInterface` object associated with the service interface type specified in `serviceInterfaceType`.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

serviceInterfaceType — Service interface type

enumerated value of `coder.descriptor.Services`

Service interface type defined in Embedded Coder Dictionary, specified as an enumerated value of `coder.descriptor.Services`. Specify the value as one of the following enumeration variants:

- `coder.descriptor.Services.SenderReceiver`
- `coder.descriptor.Services.DataTransfer`
- `coder.descriptor.Services.Timer`
- `coder.descriptor.Services.Measurement`
- `coder.descriptor.Services.ParameterTuning`

Output Arguments

serviceInterfaceTypeObj — Service interface object associated with service interface type

`coder.descriptor.SenderReceiverServiceInterface` object |
`coder.descriptor.DataTransferServiceInterface` object |
`coder.descriptor.TimerServiceInterface` object |
`coder.descriptor.MeasurementServiceInterface` object |
`coder.descriptor.ParameterTuningServiceInterface` object

Service interface object associated with the type of service interface with corresponding properties, returned as:

Input argument: <code>serviceInterfaceType</code>	Output argument: <code>serviceInterfaceTypeObj</code>
<code>coder.descriptor.Services.SenderReceiver</code>	<code>coder.descriptor.SenderReceiverServiceInterface</code>
<code>coder.descriptor.Services.DataTransfer</code>	<code>coder.descriptor.DataTransferServiceInterface</code>
<code>coder.descriptor.Services.Timer</code>	<code>coder.descriptor.TimerServiceInterface</code>
<code>coder.descriptor.Services.Measurement</code>	<code>coder.descriptor.MeasurementServiceInterface</code>
<code>coder.descriptor.Services.ParameterTuning</code>	<code>coder.descriptor.ParameterTuningServiceInterface</code>

To return a service interface object for the parameter argument tuning service interface type, pass `coder.descriptor.Services.ParameterTuning` to `getServiceInterface`, and then pass the output to `getParameterArgumentInterfaces`.

Version History

Introduced in R2022b

See Also

Embedded Coder Dictionary | Service Interfaces | `coder.descriptor.ServiceInterface` | `coder.descriptor.SenderReceiverServiceInterface` | `coder.descriptor.DataTransferServiceInterface` | `coder.descriptor.TimerServiceInterface` | `coder.descriptor.MeasurementServiceInterface` | `coder.descriptor.ParameterTuningServiceInterface` | `getParameterArgumentInterfaces`

Topics

“Get Metadata About Service Interface”

getServicesHeaderFileName

Class: `coder.descriptor.ServiceInterface`

Namespace: `coder.descriptor`

Return name of header file that contains service function declarations

Syntax

```
serviceHeaderFile = getServicesHeaderFileName(serviceObj)
```

Description

`serviceHeaderFile = getServicesHeaderFileName(serviceObj)` returns the file name that is used by the code generator to write service function declarations for the service interface `serviceObj`.

Input Arguments

serviceObj — Service interface

`coder.descriptor.ServiceInterface` object

Service interface, also referred to as the code configuration object, for which you want to retrieve the information about generated code, specified as a `coder.descriptor.ServiceInterface` object.

Output Arguments

serviceHeaderFile — Service header file name

character vector | string scalar

Service header file name that contains the service function declarations, returned as character vector or string scalar.

Version History

Introduced in R2022b

See Also

`coder.descriptor.ServiceInterface` | `getServices`

Topics

“Get Metadata About Service Interface”

coder.Dictionary Class

Namespace: coder

Configure Embedded Coder Dictionary

Description

An object of the `coder.Dictionary` class represents an Embedded Coder Dictionary. Use the object to perform operations on the Embedded Coder Dictionary, such as load packages of definitions and gain access to sections of the dictionary.

A `coder.Dictionary` object contains `coder.dictionary.Section` objects, which represent the categories of an Embedded Coder Dictionary. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects, which represent the definitions stored in that section of the Embedded Coder Dictionary. The name of the section identifies the type of code definitions that the section contains. To access the sections of an Embedded Coder Dictionary, use a `coder.Dictionary` object. To access the dictionary entries within the section, use a `coder.dictionary.Section` object.

Creation

The functions `coder.dictionary.create` and `coder.dictionary.open` create a `coder.Dictionary` object.

Methods

Public Methods

<code>getCodeInterfaceType</code>	Determine whether <code>coder.Dictionary</code> object represents a service interface configuration or data interface configuration
<code>getSection</code>	Return <code>coder.dictionary.Section</code> object that represents Embedded Coder Dictionary section
<code>getSections</code>	Return <code>coder.dictionary.Section</code> objects of an Embedded Coder Dictionary
<code>getDictionaryDefault</code>	Return default service for service interface section or default property value for data interface category
<code>setDictionaryDefault</code>	Set default service for service interface section or default property value for data interface category
<code>loadPackage</code>	Load package of code definitions into Embedded Coder Dictionary
<code>unloadPackage</code>	Unload package of code definitions from Embedded Coder Dictionary
<code>getLoadedPackages</code>	Return code definition packages loaded into Embedded Coder Dictionary
<code>refreshPackage</code>	Refresh package definitions in Embedded Coder Dictionary
<code>resetToDefault</code>	Restore Embedded Coder Dictionary to default state
<code>get</code>	Get property value of Embedded Coder Dictionary with service interface configuration
<code>set</code>	Set property value for Embedded Coder Dictionary with service interface configuration
<code>valid</code>	Determine whether <code>coder.Dictionary</code> object represents a valid Embedded Coder Dictionary

empty Determine whether Embedded Coder Dictionary is empty

Examples

Create Embedded Coder Dictionary in Simulink Data Dictionary

Create a data dictionary to contain an Embedded Coder Dictionary.

```
dataDictionary = Simulink.data.dictionary.create("DataDictionary.sldd");
```

Create an Embedded Coder Dictionary in the data dictionary and use a `coder.Dictionary` object to represent the Embedded Coder Dictionary.

```
coderDictionary = coder.dictionary.create(dataDictionary)
```

```
coderDictionary =
```

```
Dictionary with Sections:
```

```
StorageClasses: [1x1 coder.dictionary.Section]  
MemorySections: [1x1 coder.dictionary.Section]  
FunctionCustomizationTemplates: [1x1 coder.dictionary.Section]
```

The Embedded Coder Dictionary contains three `coder.dictionary.Section` objects, each of which represents a section of the dictionary.

Create a Service Interface Configuration

Create a data dictionary to contain the Embedded Coder Dictionary.

```
dataDictionary = Simulink.data.dictionary.create("DataDictionary.sldd");
```

Create an Embedded Coder Dictionary in the data dictionary and use a `coder.Dictionary` object to represent the Embedded Coder Dictionary. Specify that the interface configuration type is service interface.

```
coderDictionary = coder.dictionary.create(dataDictionary,"ServiceInterface")
```

```
coderDictionary =
```

```
Dictionary with properties and Sections:
```

```
ServicesHeaderFileName: 'services.h'  
InitTermFunctions: [1x1 coder.dictionary.Section]  
PeriodicAperiodicFunctions: [1x1 coder.dictionary.Section]  
DataReceiverInterfaces: [1x1 coder.dictionary.Section]  
DataSenderInterfaces: [1x1 coder.dictionary.Section]  
DataTransferInterfaces: [1x1 coder.dictionary.Section]  
TimerInterfaces: [1x1 coder.dictionary.Section]  
ParameterTuningInterfaces: [1x1 coder.dictionary.Section]  
ParameterArgumentTuningInterfaces: [1x1 coder.dictionary.Section]  
MeasurementInterfaces: [1x1 coder.dictionary.Section]  
SubcomponentInitTermFunctions: [1x1 coder.dictionary.Section]  
SubcomponentPeriodicAperiodicFunctions: [1x1 coder.dictionary.Section]  
SharedUtilityFunctions: [1x1 coder.dictionary.Section]  
InternalData: [1x1 coder.dictionary.Section]  
Constants: [1x1 coder.dictionary.Section]  
StorageClasses: [1x1 coder.dictionary.Section]  
DataMemorySections: [1x1 coder.dictionary.Section]  
FunctionMemorySections: [1x1 coder.dictionary.Section]
```

The Embedded Coder Dictionary contains `coder.dictionary.Section` objects, each of which represents a category of service interfaces, functions, or code definitions in the dictionary.

Version History

Introduced in R2019b

R2024a: Not measured and Not tunable as dictionary default

You can now specify "Not measured" as the default value of the `MeasurementInterfaces` section in the coder dictionary, and you can specify "Not tunable" as the default value of the `ParameterTuningInterfaces` and `ParameterArgumentTuningInterfaces` sections in the dictionary. The "Not measured" and "Not tunable" entries are included in the dictionary, and are preselected as default values in newly created dictionaries. You can select other entries as dictionary default for these sections, but you cannot delete these entries from the dictionary.

R2024a: Changes in service interfaces dictionary sections

- `Constants` and `InternalData` are new dictionary sections.
- The dictionary sections `SubcomponentInitTermFunctions` and `SubcomponentPeriodicAperiodicFunctions` replace the section `SubcomponentEntryFunctions`.

R2023b: Support for service interface configurations

Access the `coder.Dictionary` objects for Embedded Coder Dictionaries that use the service interface configuration. Previously, you could only access the `coder.Dictionary` object for an Embedded Coder Dictionary that used the data interface configuration. The `coder.Dictionary` class now supports dictionaries of both interface configuration types.

See Also

`coder.dictionary.Section` | `coder.dictionary.Entry` | Embedded Coder Dictionary

coder.dictionary.copy

Namespace: `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`. The source and destination dictionaries must use the same code interface configuration type.

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 Code Interface Configuration Between Models”. To copy individual code definitions, use the Embedded Coder Dictionary dialog box or `copyEntry`. For general information about Embedded Coder Dictionaries and code generation definitions, see “Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture”.

Examples

Copy Code Definitions from Model to Model

In the Embedded Coder Dictionary of the example model `RollAxisAutopilot`, create a storage class. Then, copy the storage class to the Embedded Coder Dictionary in `EmbeddedCoderIntro`.

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings.

```
newEntry = addEntry(storageClassesSect, 'MyStorageClass')
```

```
newEntry =
```

```
    Entry with properties:
```

```
Name: 'MyStorageClass'
DataSource: 'RollAxisAutopilot'
```

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

Open the other model, `EmbeddedCoderIntro`.

```
openExample('EmbeddedCoderIntro')
```

Copy the contents of the Embedded Coder Dictionary in `RollAxisAutopilot` to the Embedded Coder Dictionary in `EmbeddedCoderIntro`.

```
coder.dictionary.copy('RollAxisAutopilot','EmbeddedCoderIntro')
```

Open the Embedded Coder Dictionary for `EmbeddedCoderIntro`.

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

Input Arguments

sourceName — Source model file or data dictionary

character vector | string scalar

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

- 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 | string scalar

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

- 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.
- The interface configuration type of the destination dictionary must match the type of the source dictionary. Check the type by using the `getCodeInterfaceType` method.

Example: `'myLoadedModel'`

Example: 'myDictionary.slidd'

Data Types: char

Version History

Introduced in R2018a

R2023b: Support for service interface configurations

Copy code interface definitions between Embedded Coder Dictionaries that use the service interface configuration. You can still copy code definitions between Embedded Coder Dictionaries that use the data interface configuration.

See Also

Embedded Coder Dictionary | `coder.dictionary.Entry` | `copyEntry`

Topics

"Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture"

"Configure Default C Code Generation for Categories of Data Elements and Functions"

coder.dictionary.create

Namespace: coder.dictionary

Create Embedded Coder Dictionary and coder.Dictionary object

Syntax

```
coderDict = coder.dictionary.create(sourceName)
coderDict = coder.dictionary.create(sourceName,interfaceType)
```

Description

`coderDict = coder.dictionary.create(sourceName)` creates an Embedded Coder Dictionary in the model or Simulink data dictionary identified by `sourceName`. The function returns a `coder.Dictionary` object that represents the new Embedded Coder Dictionary. The Embedded Coder Dictionary uses the data interface configuration type.

When the source model or data dictionary already has an Embedded Coder Dictionary, use `coder.dictionary.open` to access the `coder.Dictionary` object. An Embedded Coder Dictionary is created when you open a model in the Embedded Coder app or when you open the Embedded Coder Dictionary dialog box for a model or a data dictionary.

`coderDict = coder.dictionary.create(sourceName,interfaceType)` creates an Embedded Coder Dictionary that uses the specified code interface type. You can create an Embedded Coder Dictionary that uses either the data interface configuration or service interface configuration by using this syntax.

Examples

Create Embedded Coder Dictionary in Simulink Data Dictionary

Create a data dictionary.

```
dataDictionary = Simulink.data.dictionary.create("DataDictionary.sldd");
```

Create an Embedded Coder Dictionary in the data dictionary.

```
coderDictionary = coder.dictionary.create(dataDictionary);
```

```
coderDictionary =
```

```
    Dictionary with Sections:
```

```
        StorageClasses: [1x1 coder.dictionary.Section]
        MemorySections: [1x1 coder.dictionary.Section]
        FunctionCustomizationTemplates: [1x1 coder.dictionary.Section]
```

Create Service Interface Configuration

Create a data dictionary to store the Embedded Coder Dictionary.

```
dataDictionary = Simulink.data.dictionary.create("DataDictionary.sldd");
```

Create an Embedded Coder Dictionary in the data dictionary. Specify the service interface configuration type.

```
coderDictionary = coder.dictionary.create(dataDictionary, "ServiceInterface");
```

```
coderDictionary =
```

```
Dictionary with properties and Sections:
```

```
    ServicesHeaderFileName: 'services.h'
    InitTermFunctions: [1x1 coder.dictionary.Section]
    PeriodicAperiodicFunctions: [1x1 coder.dictionary.Section]
    DataReceiverInterfaces: [1x1 coder.dictionary.Section]
    DataSenderInterfaces: [1x1 coder.dictionary.Section]
    DataTransferInterfaces: [1x1 coder.dictionary.Section]
    TimerInterfaces: [1x1 coder.dictionary.Section]
    ParameterTuningInterfaces: [1x1 coder.dictionary.Section]
    ParameterArgumentTuningInterfaces: [1x1 coder.dictionary.Section]
    MeasurementInterfaces: [1x1 coder.dictionary.Section]
    SubcomponentInitTermFunctions: [1x1 coder.dictionary.Section]
    SubcomponentPeriodicAperiodicFunctions: [1x1 coder.dictionary.Section]
    SharedUtilityFunctions: [1x1 coder.dictionary.Section]
    InternalData: [1x1 coder.dictionary.Section]
    Constants: [1x1 coder.dictionary.Section]
    StorageClasses: [1x1 coder.dictionary.Section]
    DataMemorySections: [1x1 coder.dictionary.Section]
    FunctionMemorySections: [1x1 coder.dictionary.Section]
```

Now you can access the sections of the dictionary and use the sections to add and edit service interface definitions.

Input Arguments

sourceName — Target model file or data dictionary

character vector | string scalar | Simulink.data.Dictionary object

Name of target model file or data dictionary, specified as a character vector, string scalar, or Simulink.data.Dictionary object.

- 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

interfaceType — Code interface configuration type

"DataInterface" (default) | "ServiceInterface"

Code interface configuration type, specified as either "DataInterface" or "ServiceInterface". If sourceName is a Simulink model, the interfaceType must be "DataInterface".

Choose the code interface configuration type by considering how you will deploy the generated code:

- Data interface configuration — The code generator creates a program intended to run on the target device. The generated code uses the data interfaces that you define.
- Service interface configuration — The code generator creates an algorithm that you intend to deploy within a larger application. The platform middleware calls the algorithm code and provides the services for the target device. The generated code calls the services according to your definitions in the service interface configuration. To create a service interface configuration, you must use an Embedded Code Dictionary that is stored in a Simulink data dictionary.

Example: "ServiceInterface"

Data Types: char

Output Arguments

coderDict — New Embedded Code Dictionary

`coder.Dictionary` object

Newly created Embedded Code Dictionary, returned as a `coder.Dictionary` object.

Version History

Introduced in R2019b

R2024a: Not measured and Not tunable as dictionary default

You can now specify "Not measured" as the default value of the `MeasurementInterfaces` section in the code dictionary, and you can specify "Not tunable" as the default value of the `ParameterTuningInterfaces` and `ParameterArgumentTuningInterfaces` sections in the dictionary. The "Not measured" and "Not tunable" entries are included in the dictionary, and are preselected as default values in newly created dictionaries. You can select other entries as dictionary default for these sections, but you cannot delete these entries from the dictionary.

R2024a: Changes in service interfaces dictionary sections

- Constants and `InternalData` are new dictionary sections.
- The dictionary sections `SubcomponentInitTermFunctions` and `SubcomponentPeriodicAperiodicFunctions` replace the section `SubcomponentEntryFunctions`.

R2023b: Support for service interface configurations

Create an Embedded Code Dictionary that uses the service interface configuration by specifying "ServiceInterface" for the second input argument. You can still create an Embedded Code Dictionary that uses the data interface configuration by specifying "DataInterface" or by not specifying the second input argument.

See Also

`coder.Dictionary` | Embedded Code Dictionary

Topics

“Create Data Interface Configuration Programmatically”

coder.dictionary.exist

Namespace: coder.dictionary

Determine whether Embedded Coder Dictionary exists in model or data dictionary

Syntax

```
tf = coder.dictionary.exist(sourceName)
```

Description

`tf = coder.dictionary.exist(sourceName)` returns true if a model or Simulink data dictionary `sourceName` contains an Embedded Coder Dictionary.

An Embedded Coder Dictionary is created when you open a model in the Embedded Coder app or when you open the Embedded Coder Dictionary dialog box for a model or a data dictionary.

Examples

Create an Embedded Coder Dictionary in a Data Dictionary

Create a Simulink data dictionary. Then, determine if the data dictionary contains an Embedded Coder Dictionary. Use the result to create an Embedded Coder Dictionary or to open the existing dictionary.

Create a Simulink data dictionary.

```
dataDictionary = Simulink.data.dictionary.create('DataDictionary.slidd');
```

Determine whether the data dictionary contains an Embedded Coder Dictionary. Open the existing Embedded Coder Dictionary or create and open an Embedded Coder Dictionary in the data dictionary.

```
if coder.dictionary.exist('DataDictionary.slidd')
    myCoderDict = coder.dictionary.open('DataDictionary.slidd');
else
    myCoderDict = coder.dictionary.create('DataDictionary.slidd');
end
```

Because the data dictionary did not contain an Embedded Coder Dictionary, the code creates a coder dictionary in the data dictionary.

Input Arguments

sourceName — Name of model or Simulink data dictionary

character vector | string scalar

Name of the model or Simulink data dictionary, specified as a character vector or string scalar.

Example: 'RollAxisAutopilot'

Output Arguments

tf — True or false result

1 | 0 | logical array

True or false result, returned as 1 or 0 of data type `logical`.

Version History

Introduced in R2020b

See Also

`coder.Dictionary` | `coder.dictionary.create` | `coder.dictionary.open` | `coder.dictionary.remove`

Topics

“Create Data Interface Configuration Programmatically”

coder.dictionary.move

Namespace: coder.dictionary

Migrate code generation definitions between models and data dictionaries

Syntax

```
coder.dictionary.move(sourceName,destinationName)
```

Description

`coder.dictionary.move(sourceName,destinationName)` moves code generation definitions, such as storage classes, from the Embedded Code Dictionary in `sourceName` to the Embedded Code Dictionary in `destinationName`. The definitions are removed from `sourceName`. The source and destination dictionaries must use the same code interface configuration type. To copy code definitions from one Embedded Code 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 more information, see “Share Code Interface Configuration Between Models”.
- In a hierarchy of referenced Simulink data dictionaries, move the Embedded Code Dictionary from one data dictionary to another. In a hierarchy of referenced dictionaries, only one dictionary can store an Embedded Code Dictionary.

For general information about Embedded Code Dictionaries and code generation definitions, see “Define Service Interfaces, 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 Code Dictionary of the example model `RollAxisAutopilot`. Then, move the storage class to the Embedded Code Dictionary in `EmbeddedCoderIntro`.

Open the model `RollAxisAutopilot` and represent the Embedded Code Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings.

```
newEntry = addEntry(storageClassesSect, 'MyStorageClass')
```

```
newEntry =
```

```
    Entry with properties:
```

```
        Name: 'MyStorageClass'  
    DataSource: 'RollAxisAutopilot'
```

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

Open the other model, `EmbeddedCoderIntro`.

```
openExample('EmbeddedCoderIntro')
```

Move the contents of the Embedded Coder Dictionary in `RollAxisAutopilot` to the Embedded Coder Dictionary in `EmbeddedCoderIntro`.

```
coder.dictionary.move('RollAxisAutopilot', 'EmbeddedCoderIntro')
```

Open the Embedded Coder Dictionary for `EmbeddedCoderIntro`.

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

Input Arguments

sourceName — Source model file or data dictionary

character vector | string scalar

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

- 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 | string scalar

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

- 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.
- The interface configuration type of the destination dictionary must match the type of the source dictionary. Check the type by using the `getCodeInterfaceType` method.

Example: `'myLoadedModel'`

Example: `'myDictionary.sldd'`

Data Types: `char`

Version History

Introduced in R2018a

R2023b: Support for service interface configurations

Move code interface definitions between Embedded Coder Dictionaries that use the service interface configuration. You can still move code definitions between Embedded Coder Dictionaries that use the data interface configuration.

See Also

Embedded Coder Dictionary

Topics

“Migrate Data Interface Configuration from Model File to Shared Data Dictionary”

“Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture”

“Configure Default C Code Generation for Categories of Data Elements and Functions”

coder.dictionary.open

Namespace: `coder.dictionary`

Open Embedded Coder Dictionary object

Syntax

```
coderDictionaryObj = coder.dictionary.open(sourceName)
```

Description

`coderDictionaryObj = coder.dictionary.open(sourceName)` opens the Embedded Coder Dictionary in the model or Simulink data dictionary identified by `sourceName`. The function returns a `coder.Dictionary` object representing the Embedded Coder Dictionary.

Examples

Open Existing Embedded Coder Dictionary

Open the model `RollAxisAutopilot`, which has an Embedded Coder Dictionary.

```
openExample('RollAxisAutopilot')
```

Open the Embedded Coder Dictionary and represent it with a `coder.Dictionary` object.

```
coderDictionaryObj = coder.dictionary.open('RollAxisAutopilot')
```

```
coderDictionaryObj =
```

```
    Dictionary with Sections:
```

```
                StorageClasses: [1x1 coder.dictionary.Section]
                MemorySections: [1x1 coder.dictionary.Section]
FunctionCustomizationTemplates: [1x1 coder.dictionary.Section]
```

Input Arguments

sourceName — Target model file or data dictionary

character vector | string scalar | `Simulink.data.Dictionary` object

Name of the model file or data dictionary that contains the Embedded Coder 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.slidd'

Data Types: char

Output Arguments

coderDictionaryObj — Embedded Coder Dictionary

`coder.Dictionary` object

Embedded Coder Dictionary, returned as a `coder.Dictionary` object.

Version History

Introduced in R2019b

See Also

`coder.Dictionary` | Embedded Coder Dictionary

coder.dictionary.remove

Namespace: `coder.dictionary`

Remove Embedded Coder Dictionary from model or Simulink data dictionary

Syntax

`remove(sourceName)`

Description

`remove(sourceName)` removes Embedded Coder Dictionary definitions from your 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 does not have an attached 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 23-0) or open the Code perspective for a model, Simulink creates an Embedded Coder Dictionary in the model file. In this example, create a code generation definition (a storage class) in the Embedded Coder Dictionary of the example model `RollAxisAutopilot`, then remove the Embedded Coder Dictionary from the model.

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings.

```
newEntry = addEntry(storageClassesSect, 'MyStorageClass')
```

```
newEntry =
```

```
    Entry with properties:
```

```
        Name: 'MyStorageClass'
    DataSource: 'RollAxisAutopilot'
```

At the command prompt, remove the Embedded Coder Dictionary from the model.

```
coder.dictionary.remove('RollAxisAutopilot')
```

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 | string scalar

Target model file or data dictionary, specified as a character vector or string scalar.

- 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.

Version History

Introduced in R2018a

R2019a: For models, local Embedded Coder Dictionary is not removed

Behavior changed in R2019a

When you use the function `coder.dictionary.remove` on a model, the function removes only the local definitions from packages that you have loaded and local custom definitions. The model still contains the local dictionary with definitions from the `SimulinkBuiltIn` package.

See Also

Embedded Coder Dictionary

Topics

“Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture”

“Configure Default C Code Generation for Categories of Data Elements and Functions”

copyEntry

Class: `coder.dictionary.Section`

Namespace: `coder.dictionary`

Copy Embedded Coder Dictionary entry

Syntax

```
newEntry = copyEntry(sectionObj,defName)
```

```
newEntry = copyEntry(sectionObj,defName,targetCoderDict)
```

Description

`newEntry = copyEntry(sectionObj,defName)` copies the definition named `defName` in the Embedded Coder dictionary section represented by `sectionObj`. The `copyEntry` syntax returns a `coder.dictionary.Entry` object that represents the new definition.

`newEntry = copyEntry(sectionObj,defName,targetCoderDict)` creates a copy of the definition in the target Embedded Coder Dictionary represented by `targetCoderDict`.

Input Arguments

sectionObj — Source Embedded Coder Dictionary section

`coder.dictionary.Section` object

Source Embedded Coder Dictionary section that contains the code definition, specified as a `coder.dictionary.Section` object. The section name identifies the type of code definition that `addEntry` creates.

defName — Name of Embedded Coder Dictionary definition

character vector | string scalar

Name of Embedded Coder Dictionary definition that you want to copy, specified as a string.

Example: 'StorageClass2'

targetCoderDict — Target Embedded Coder Dictionary

`coder.Dictionary` object

Target Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Output Arguments

newEntry — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

New Embedded Coder Dictionary entry, returned as a `coder.dictionary.Entry` object. The new entry represents a copy of the code definition in the target section of the Embedded Coder Dictionary.

Examples

Copy Storage Class in Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. The dictionary contains an example storage class definition named `SignalStruct`.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Copy the storage class definition `SignalStruct`. The new storage class has the name `SignalStruct_copy`.

```
newEntry = copyEntry(storageClassesSect, 'SignalStruct')
```

```
newEntry =
```

```
    Entry with properties:
```

```
        Name: 'SignalStruct_copy'
    DataSource: 'RollAxisAutopilot'
```

Copy Storage Class to Different Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary with a `coder.Dictionary` object. The dictionary contains an example storage class definition named `SignalStruct`.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Create a data dictionary that contains an Embedded Coder Dictionary to store the copy of `SignalStruct`.

```
newSLDD = Simulink.data.dictionary.create('newSLDD.sldd');
slddCoderDictionary = coder.dictionary.create('newSLDD.sldd');
```

Copy the storage class definition `SignalStruct` from the model's Embedded Coder Dictionary to the data dictionary. The new storage class has the name `SignalStruct`.

```
newEntry = copyEntry(storageClassesSect, 'SignalStruct', slddCoderDictionary)
```

```
newEntry =
```


Entry with properties:

Name: 'SignalStruct'
DataSource: 'X:\newSLDD.sldd'

Version History

Introduced in R2019b

See Also

`coder.dictionary.Entry` | `coder.Dictionary`

empty

Class: `coder.Dictionary`

Namespace: `coder`

Determine whether Embedded Coder Dictionary is empty

Syntax

```
tf = empty(coderDictionaryObj)
```

Description

`tf = empty(coderDictionaryObj)` returns `true` if the `coder.Dictionary` object `coderDictionaryObj` is empty. When you remove the Embedded Coder Dictionary from a model or a data dictionary by using `coder.dictionary.remove`, the Embedded Coder Dictionary becomes empty.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Output Arguments

tf — True or false result

1 | 0 | logical array

True or false result, returned as a 1 or 0 of data type `logical`.

Examples

Check If Embedded Coder Dictionary Object Is Empty

Create a data dictionary named '`dataDict.sldd`' in the base workspace, and then create an Embedded Coder Dictionary in the data dictionary. In the base workspace, represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `myCoderDictObj`.

```
Simulink.data.dictionary.create('dataDict.sldd');  
myCoderDictObj = coder.dictionary.create('dataDict.sldd');
```

Check if the Embedded Coder Dictionary is empty. When you created the Embedded Coder Dictionary, definitions from the Simulink package were loaded into it, so the dictionary is initially not empty.

```
empty(myCoderDictObj)
```

```
ans =  
    logical  
    0
```

Remove the Embedded Coder Dictionary from the data dictionary and again check to see if it is empty.

```
coder.dictionary.remove('dataDict.sldd');  
empty(myCoderDictObj)
```

```
ans =  
    logical  
    1
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary`

get

Class: `coder.Dictionary`

Namespace: `coder`

Get property value of Embedded Coder Dictionary with service interface configuration

Syntax

```
value = get(dictionary,property)
```

Description

`value = get(dictionary,property)` returns the value of the specified property for the Embedded Coder Dictionary. `get` supports only Embedded Coder Dictionaries that use the service interface configuration.

Input Arguments

dictionary — Embedded Coder Dictionary

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

property — Property of `coder.Dictionary` object

character vector | string scalar

Property of the `coder.Dictionary` object, specified as a character vector or string scalar.

Example: `'ServicesHeaderFileName'`

Data Types: `string`

Output Arguments

value — Value of `coder.Dictionary` object

string scalar

Value of the `coder.Dictionary` property, returned as a string scalar.

Examples

Determine Header File for Service Interfaces

Create a Simulink data dictionary that contains an Embedded Coder Dictionary. Create the Embedded Coder Dictionary so that it represents a service interface configuration.

```
dictionaryFile = Simulink.data.dictionary.create('codeDefinitions.sldd');  
codeDictionary = coder.dictionary.create('codeDefinitions.sldd','ServiceInterface');
```

Get the name of the header file that the service interfaces use.

```
headerFile = get(codeDictionary, 'ServicesHeaderFileName')  
headerFile =  
    'services.h'
```

The Embedded Coder Dictionary uses the default header file `services.h` for the services.

Version History

Introduced in R2023b

See Also

`coder.Dictionary`

getCodeInterfaceType

Class: `coder.Dictionary`

Namespace: `coder`

Determine whether `coder.Dictionary` object represents a service interface configuration or data interface configuration

Syntax

```
interfaceType = getCodeInterfaceType(dict)
```

Description

`interfaceType = getCodeInterfaceType(dict)` returns the code interface type of the `coder.Dictionary` object. The type is either `ServiceInterface` or `DataInterface`. The interface type determines which types of code definitions you use in the dictionary and how you can deploy the generated code.

- **Data interface** — The code generator creates a program intended to run on the target device. You define the code interfaces by using storage classes, memory sections, and function customization templates.
- **Service interface** — The code generator creates an algorithm that you intend to deploy within a larger application for a target platform. The platform middleware calls the algorithm code and defines services for the target device. You define how the generated code calls the services by creating function and service interface definitions in the dictionary.

Input Arguments

dict — Embedded Coder Dictionary

`coder.Dictionary` object

Embedded Coder dictionary, specified as a `coder.Dictionary` object.

Output Arguments

interfaceType — Code interface configuration type

`'ServiceInterface' | 'DataInterface'`

Code interface configuration type of the coder dictionary, returned as a string. The interface type is either `'ServiceInterface'` or `'DataInterface'`.

Examples

Determine Interface Type of Embedded Coder Dictionary

Open the model `RollAxisAutopilot`. Assign the `coder.Dictionary` object for the model to the variable `coderDictObj`.

```
openExample('RollAxisAutopilot')  
coderDictObj = coder.dictionary.open('RollAxisAutopilot');
```

Determine the code interface type for the Embedded Coder Dictionary by using the `coder.Dictionary` object.

```
interfaceType = getCodeInterfaceType(coderDictObj)  
  
interfaceType =  
  
    'DataInterface'
```

The dictionary has the data interface type. For the data interface type, you can add storage classes, memory sections, and function customization templates to represent the code interface.

Version History

Introduced in R2023b

See Also

`coder.Dictionary`

Topics

“Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture”

getDictionaryDefault

Class: coder.Dictionary

Namespace: coder

Return default service for service interface section or default property value for data interface category

Syntax

```
defaultEntry = getDictionaryDefault(serviceDict,sectionName)
defaultPropVal = getDictionaryDefault(dataDict,categoryName,propName)
```

Description

`defaultEntry = getDictionaryDefault(serviceDict,sectionName)` returns the default entry of the service interface section `sectionName` in the service interfaces coder dictionary `serviceDict`.

`defaultPropVal = getDictionaryDefault(dataDict,categoryName,propName)` returns the default value of the property `propName` of the category `categoryName` in the data interfaces coder dictionary `dataDict`.

Input Arguments

serviceDict — Embedded Coder Dictionary with service interfaces configuration

`coder.Dictionary` object

Embedded Coder Dictionary that contains a service interface configuration, specified as a `coder.Dictionary` object.

sectionName — Service interface section name

"DataReceiverInterfaces" | "DataSenderInterfaces" | ...

Service interface section name, specified as a string scalar or character vector that corresponds to a section of the service interface definitions in the dictionary. Sections include:

- "DataReceiverInterfaces"
- "DataSenderInterfaces"
- "DataTransferInterfaces"
- "InitTermFunctions"
- "MeasurementInterfaces"
- "ParameterArgumentTuningInterfaces"
- "ParameterTuningInterfaces"
- "PeriodicAperiodicFunctions"
- "SharedUtilityFunctions"
- "SubcomponentInitTermFunctions"

- "SubcomponentPeriodicAperiodicFunctions"
- "TimerInterfaces"

dataDict — Embedded Coder Dictionary with data interfaces configuration

`coder.Dictionary` object

Embedded Coder Dictionary that contains a data interface configuration, specified as a `coder.Dictionary` object.

categoryName — Data interface category name

"InitializeTerminate" | "Execution" | "SharedUtility" | "Inports" | "Outports" | ...

Data interface category name, specified as a string scalar or character vector that corresponds to a category in one of the category groups in the dictionary, Function Defaults and Data Defaults.

Function Defaults categories are:

- "InitializeTerminate"
- "Execution"
- "SharedUtility"

Data Defaults categories are:

- "Inports"
- "Outports"
- "ModelParameters"
- "ModelParameterArguments"
- "ExternalParameters"
- "SharedLocalDataStores"
- "GlobalDataStores"
- "InternalData"
- "Constants"

propName — Name of category property

"FunctionCustomizationTemplate" | "StorageClass"

Name of category property for which to retrieve the dictionary default value, specified as a string scalar or character vector. The list of available properties to use with this method depends on the category type and its specific configuration.

For categories within Function Defaults, the `FunctionCustomizationTemplate` property is available for use with this method. When the `FunctionCustomizationTemplate` is set to `Default`, an additional property, `MemorySection`, becomes available for use with this method for these categories.

Data Defaults categories have the `StorageClass` property available for use with this method. The set of available properties is dynamic and varies based on the assigned value of the `StorageClass` property. For example, when the `StorageClass` property is assigned the value `ExportToFile`, which is allowed for certain categories, these additional properties become available for use with this method:

- "HeaderFile"
- "DefinitionFile"
- "Owner"
- "PreserveDimensions"

To learn more about storage classes and their properties, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Output Arguments

defaultEntry — Default entry

`coder.dictionary.Entry` object

Default entry for the specified service interface section, returned as a `coder.dictionary.Entry` object.

defaultPropVal — Default property value

character vector

Default property value, returned as a character vector.

Examples

Determine Default Entry for Measurement Interfaces

Create a Simulink data dictionary that contains an Embedded Coder Dictionary. Create the Embedded Coder Dictionary so that it represents a service interface configuration.

```
dictionaryFile = Simulink.data.dictionary.create("codeDefinitions.sldd");  
codeDictionary = coder.dictionary.create("codeDefinitions.sldd", "ServiceInterface");
```

Get the default measurement interface entry.

```
meas = getDictionaryDefault(codeDictionary, "MeasurementInterfaces")
```

```
meas =
```

Entry with properties:

```
      Name: 'MeasurementExample1'  
DataSource: 'C:\work\serviceDefinitions.sldd'  
StorageClass: 'MeasurementStruct'
```

For this example, the default measurement interface is `MeasurementExample1`.

Get and Set Default Outports Properties in Data Interface Coder Dictionary

Create a Simulink® data dictionary, then create the Embedded Coder Dictionary to represent a data interface configuration.

```
dictFileName = "dataDict.sldd";  
dataDict = Simulink.data.dictionary.create(dictFileName);  
coderDataDict = coder.dictionary.create(dataDict, "DataInterface");
```

Use the `getDictionaryDefault` method to see the default `StorageClass` of the `Outputs` category.

```
getDictionaryDefault(coderDataDict,"Outputs","StorageClass")
```

```
ans =  
'Default'
```

Set the default `StorageClass` of the `Outputs` category to `GetSet`. Then specify `GetFunction` (the naming rule of the get function) as `get_outports_$N` and specify `SetFunction` (the naming rule of the set function) as `set_outports_$N`.

```
setDictionaryDefault(coderDataDict,"Outputs","StorageClass","GetSet")  
setDictionaryDefault(coderDataDict,"Outputs","GetFunction","get_outports_$N")  
setDictionaryDefault(coderDataDict,"Outputs","SetFunction","set_outports_$N")
```

Discard and close the dictionary, then delete the dictionary file. This allows you to run this example more than once without getting an error while attempting to create the dictionary.

```
discardChanges(dataDict)  
Simulink.data.dictionary.closeAll("-save")  
delete(dictFileName)
```

Version History

Introduced in R2023b

R2024a: Changes in service interfaces dictionary sections

Behavior changed in R2024a

The dictionary sections `SubcomponentInitTermFunctions` and `SubcomponentPeriodicAperiodicFunctions` replace the section `SubcomponentEntryFunctions`.

See Also

`coder.Dictionary` | `setDictionaryDefault`

getLoadedPackages

Class: `coder.Dictionary`

Namespace: `coder`

Return code definition packages loaded into Embedded Coder Dictionary

Syntax

```
packages = getLoadedPackages(coderDictionaryObj)
```

Description

`packages = getLoadedPackages(coderDictionaryObj)` lists packages loaded into the Embedded Coder Dictionary that `coderDictionaryObj` represents.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Output Arguments

packages — Packages

array of strings

Packages loaded into the Embedded Coder Dictionary, returned as an array of strings.

Examples

List Package Definitions Loaded into Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
openExample('RollAxisAutopilot')  
coderDictObj = coder.dictionary.open('RollAxisAutopilot');
```

Load the package `mpt` into the Embedded Coder Dictionary. You can configure the model to use definitions from the package.

```
loadPackage(coderDictObj, 'mpt')
```

List the packages loaded into the Embedded Coder Dictionary.

```
getLoadedPackages(coderDictObj)
```

```
ans =
```

1×2 string array

"Simulink" "mpt"

Version History

Introduced in R2024a

See Also

`coder.Dictionary`

Topics

"Refer to Code Generation Definitions in a Package" on page 23-5

getSection

Class: `coder.Dictionary`

Namespace: `coder`

Return `coder.dictionary.Section` object that represents Embedded Coder Dictionary section

Syntax

```
sectionObj = getSection(coderDict,sectionName)
```

Description

`sectionObj = getSection(coderDict,sectionName)` returns a `coder.dictionary.Section` object that represents one section, `sectionName`, of an Embedded Coder Dictionary, which `coderDict` represents. Use the section to access the code definitions of the type identified by the section name.

Input Arguments

coderDict — Embedded Coder Dictionary containing section

`coder.Dictionary` object

Embedded Coder Dictionary that contains the section you want to access, specified as a `coder.Dictionary` object.

sectionName — Name of section

`string scalar` | `character vector`

Name of the section you want to access in the Embedded Coder Dictionary, specified as a string scalar or character vector. The section name identifies the type of code definitions that the section contains. The sections in the dictionary depend on the code interface configuration type of the dictionary.

A service interface configuration contains these sections:

- `"InitTermFunctions"`
- `"PeriodicAperiodicFunctions"`
- `"DataReceiverInterfaces"`
- `"DataSenderInterfaces"`
- `"DataTransferInterfaces"`
- `"TimerInterfaces"`
- `"ParameterTuningInterfaces"`
- `"ParameterArgumentTuningInterfaces"`
- `"MeasurementInterfaces"`
- `"SubcomponentInitTermFunctions"`
- `"SubcomponentPeriodicAperiodicFunctions"`

- "SharedUtilityFunctions"
- "InternalData"
- "Constants"
- "StorageClasses"
- "DataMemorySections"
- "FunctionMemorySections"

A data interface configuration contains these sections:

- "StorageClasses"
- "MemorySections"
- "FunctionCustomizationTemplates"

Output Arguments

sectionObj — Embedded Coder Dictionary section

`coder.dictionary.Section` object

Section in the Embedded Coder Dictionary, returned as a `coder.dictionary.Section` object. The section contains `coder.dictionary.Entry` objects that represent code definitions of the type identified by `sectionName`.

Examples

Access Storage Classes in Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
openExample("RollAxisAutopilot")
coderDictObj = coder.dictionary.open("RollAxisAutopilot")
```

```
coderDictObj =
```

```
    Dictionary with Sections:
```

```
                StorageClasses: [1x1 coder.dictionary.Section]
                MemorySections: [1x1 coder.dictionary.Section]
FunctionCustomizationTemplates: [1x1 coder.dictionary.Section]
```

Represent the Storage Classes section of the Embedded Coder Dictionary by using a `coder.dictionary.Section` object named `SCSectObj`.

```
SCSectObj = getSection(coderDictObj,"StorageClasses")
```

```
SCSectObj =
```

```
    Section with properties:
```

Name: 'StorageClasses'

Version History

Introduced in R2019b

R2024a: Changes in service interfaces dictionary sections

- Constants and InternalData are new dictionary sections.
- The dictionary sections SubcomponentInitTermFunctions and SubcomponentPeriodicAperiodicFunctions replace the section SubcomponentEntryFunctions.

R2023b: Support for service interface configurations

Access the sections of Embedded Coder Dictionaries that use the service interface configuration. You can still access sections in Embedded Coder Dictionaries that use the data interface configuration.

See Also

`coder.Dictionary` | `coder.dictionary.Section`

getSections

Class: `coder.Dictionary`

Namespace: `coder`

Return `coder.dictionary.Section` objects of an Embedded Code Dictionary

Syntax

```
sections = getSections(coderDictionaryObj)
```

Description

`sections = getSections(coderDictionaryObj)` returns the `coder.dictionary.Section` objects contained in the Embedded Code Dictionary that `coderDictionaryObj` represents. Use the section objects to access code definitions of different types identified by the section names.

Input Arguments

coderDictionaryObj — Embedded Code Dictionary

`coder.Dictionary` object

Embedded Code Dictionary, specified as a `coder.Dictionary` object. Before you use this function, represent the dictionary with a `coder.Dictionary` object by using, for example, the `coder.dictionary.create` function or the `coder.dictionary.open` function.

Output Arguments

sections — Sections of Embedded Code Dictionary

array of `coder.dictionary.Section` objects

Sections of the Embedded Code Dictionary, returned as an array of `coder.dictionary.Section` objects. The sections in the dictionary depend on the code interface configuration type of the dictionary.

A service interface configuration contains these sections:

- `InitTermFunctions`
- `PeriodicAperiodicFunctions`
- `DataReceiverInterfaces`
- `DataSenderInterfaces`
- `DataTransferInterfaces`
- `TimerInterfaces`
- `ParameterTuningInterfaces`
- `ParameterArgumentTuningInterfaces`
- `MeasurementInterfaces`

- SubcomponentInitTermFunctions
- SubcomponentPeriodicAperiodicFunctions
- SharedUtilityFunctions
- InternalData
- Constants
- StorageClasses
- DataMemorySections
- FunctionMemorySections

A data interface configuration contains these sections:

- StorageClasses
- MemorySections
- FunctionCustomizationTemplates

Examples

Access Sections in Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
openExample("RollAxisAutopilot")
coderDictObj = coder.dictionary.open("RollAxisAutopilot")

coderDictObj =
```

Dictionary with Sections:

```
StorageClasses: [1x1 coder.dictionary.Section]
MemorySections: [1x1 coder.dictionary.Section]
FunctionCustomizationTemplates: [1x1 coder.dictionary.Section]
```

Access the sections of the dictionary by using an array of the corresponding `coder.dictionary.Section` objects.

```
sectionsArray = getSections(coderDictObj)

sectionsArray =
```

1x3 Section array with properties:

Name

Use each `coder.dictionary.Section` object to access the entries in that section of the dictionary. The first section contains entries that represent function customization template definitions.

```
sectObj = sectionsArray(1)

sectObj =
```

Section with properties:

Name: 'FunctionCustomizationTemplates'

Version History

Introduced in R2019b

R2024a: Changes in service interfaces dictionary sections

- Constants and InternalData are new dictionary sections.
- The dictionary sections SubcomponentInitTermFunctions and SubcomponentPeriodicAperiodicFunctions replace the section SubcomponentEntryFunctions.

R2023b: Support for service interface configurations

Access the sections of Embedded Coder Dictionaries that use the service interface configuration. You can still access sections in Embedded Coder Dictionaries that use the data interface configuration.

See Also

`coder.Dictionary` | `coder.dictionary.Section` | `coder.dictionary.Entry`

loadPackage

Class: `coder.Dictionary`

Namespace: `coder`

Load package of code definitions into Embedded Coder Dictionary

Syntax

```
loadPackage(coderDictionaryObj, pkgName)
```

Description

`loadPackage(coderDictionaryObj, pkgName)` loads the code definitions in the package `pkgName` into the Embedded Coder Dictionary that `coderDictionaryObj` represents.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

pkgName — package

character vector | string scalar

Package of code definitions to load into the Embedded Coder Dictionary.

Example: `'Simulink'`

Examples

Load Package Definitions into Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
openExample('RollAxisAutopilot')  
coderDictObj = coder.dictionary.open('RollAxisAutopilot');
```

Load the package `mpt` into the Embedded Coder Dictionary. You can configure the model to use definitions from the package.

```
loadPackage(coderDictObj, 'mpt')
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary`

Topics

“Refer to Code Generation Definitions in a Package” on page 23-5

refreshPackage

Class: `coder.Dictionary`

Namespace: `coder`

Refresh package definitions in Embedded Coder Dictionary

Syntax

```
refreshPackage(coderDictionaryObj)
```

Description

`refreshPackage(coderDictionaryObj)` refreshes code definitions stored in packages, which you create and modify by using the Custom Storage Class Designer. You do not need to refresh code definitions that you create in the Embedded Coder Dictionary. The `refreshPackage` function unloads and reloads the packages that the Embedded Coder Dictionary represented by `coderDictionaryObj` refers to. When you change the code definitions in a package, use `refreshPackage` to reload the package in each Embedded Coder Dictionary that refers to the package.

Input Arguments

`coderDictionaryObj` — Embedded Coder Dictionary object

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Examples

Refresh Package Definitions in Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
openExample('RollAxisAutopilot')  
coderDictObj = coder.dictionary.open('RollAxisAutopilot');
```

Refresh the packages in the Embedded Coder Dictionary. If you changed definitions in the packages, the changes now appear in the Embedded Coder Dictionary.

```
refreshPackage(coderDictObj)
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary`

Topics

“Change Code Generation Definitions”

resetToDefault

Class: `coder.Dictionary`

Namespace: `coder`

Restore Embedded Coder Dictionary to default state

Syntax

```
resetToDefault(coderDictionaryObj)
```

Description

`resetToDefault(coderDictionaryObj)` restores the Embedded Coder Dictionary represented by `coderDictionaryObj` to its default state. The default Embedded Coder Dictionary includes definitions only from the `Simulink` and `SimulinkBuiltIn` packages. `resetToDefault` removes custom definitions from the dictionary.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object

`coder.Dictionary` object

Embedded Coder Dictionary specified as a `coder.Dictionary` object.

Examples

Reset Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary with a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`. The Embedded Coder Dictionary for `RollAxisAutopilot` contains two example storage classes and the default storage classes.

```
openExample('RollAxisAutopilot')  
coderDictObj = coder.dictionary.open('RollAxisAutopilot');
```

Reset the Embedded Coder Dictionary. Resetting the dictionary removes the example storage classes and leaves only the definitions in the `Simulink` and `SimulinkBuiltIn` packages.

```
resetToDefault(coderDictObj)
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary`

set

Class: `coder.Dictionary`

Namespace: `coder`

Set property value for Embedded Coder Dictionary with service interface configuration

Syntax

```
set(dictionary,property,value)
```

Description

`set(dictionary,property,value)` sets the property value for the specified Embedded Coder Dictionary. `set` supports only Embedded Coder Dictionaries that use the service interface configuration.

Input Arguments

dictionary — Embedded Coder Dictionary

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

property — Property of `coder.Dictionary` object

character vector | string scalar

Property of the `coder.Dictionary` object, specified as a character vector or string scalar.

Example: `'ServicesHeaderFileName'`

Data Types: `string`

value — Value to set for property

character vector | string scalar

Value to set for the property of the `coder.Dictionary` object, specified as a character vector or string scalar.

Example: `'my_services.h'`

Data Types: `string`

Examples

Specify Header File for Service Interfaces

Create a Simulink data dictionary that contains an Embedded Coder Dictionary. Create the Embedded Coder Dictionary so that it represents a service interface configuration.

```
dictionaryFile = Simulink.data.dictionary.create('codeDefinitions.sldd');  
codeDictionary = coder.dictionary.create('codeDefinitions.sldd','ServiceInterface');
```

Get the name of the header file that the service interfaces use.

```
headerFile = get(codeDictionary, 'ServicesHeaderFileName')  
headerFile =  
    'services.h'
```

The Embedded Coder Dictionary uses the default header file `services.h` for the services.

Configure the dictionary to use the header file `my_services.h` instead.

```
set(codeDictionary, 'ServicesHeaderFileName', 'my_services.h');
```

The Embedded Coder Dictionary now uses the header file `my_services.h` for the services.

Version History

Introduced in R2023b

See Also

`coder.Dictionary`

setDictionaryDefault

Class: `coder.Dictionary`

Namespace: `coder`

Set default service for service interface section or default property value for data interface category

Syntax

```
setDictionaryDefault(serviceDict,sectionName,defaultEntry)
setDictionaryDefault(dataDict,categoryName,propName,defaultPropVal)
```

Description

`setDictionaryDefault(serviceDict,sectionName,defaultEntry)` specifies the default entry `defaultEntry` for the service interface section `sectionName` in the service interfaces coder dictionary `serviceDict`.

`setDictionaryDefault(dataDict,categoryName,propName,defaultPropVal)` specifies the default value `defaultPropVal` for the property `propName` of the category `categoryName` in the data interfaces coder dictionary `dataDict`.

Input Arguments

serviceDict — Embedded Coder Dictionary

`coder.Dictionary` object

Embedded Coder Dictionary that contains a service interface configuration, specified as a `coder.Dictionary` object.

sectionName — Service interface section name

"DataReceiverInterfaces" | "DataSenderInterfaces" | ...

Service interface section name, specified as a string scalar or character vector that corresponds to a section of the service interface definitions in the dictionary. Sections include:

- "DataReceiverInterfaces"
- "DataSenderInterfaces"
- "DataTransferInterfaces"
- "InitTermFunctions"
- "MeasurementInterfaces"
- "ParameterArgumentTuningInterfaces"
- "ParameterTuningInterfaces"
- "PeriodicAperiodicFunctions"
- "SharedUtilityFunctions"
- "SubcomponentInitTermFunctions"
- "SubcomponentPeriodicAperiodicFunctions"

- "TimerInterfaces"

defaultEntry — Service interface entry name

string scalar | character vector

The name of the service interface entry to set as default, specified as a string scalar or character vector. To get the available entries for a section, use the `find` method for the section.

Example: `find(getSection(serviceDict,"DataReceiverInterfaces")).Name`

dataDict — Embedded Coder Dictionary with data interfaces configuration

`coder.Dictionary` object

Embedded Coder Dictionary that contains a data interface configuration, specified as a `coder.Dictionary` object.

categoryName — Data interface category name

"InitializeTerminate" | "Execution" | "SharedUtility" | "Inports" | "Outports" | ...

Data interface category name, specified as a string scalar or character vector that corresponds to a category in one of the category groups in the dictionary, `Function Defaults` and `Data Defaults`.

`Function Defaults` categories are:

- "InitializeTerminate"
- "Execution"
- "SharedUtility"

`Data Defaults` categories are:

- "Inports"
- "Outports"
- "ModelParameters"
- "ModelParameterArguments"
- "ExternalParameters"
- "SharedLocalDataStores"
- "GlobalDataStores"
- "InternalData"
- "Constants"

propName — Name of category property

"FunctionCustomizationTemplate" | "StorageClass"

Name of category property for which to set the dictionary default value, specified as a string scalar or character vector. The list of available properties to use with this method depends on the category type and its specific configuration.

For categories within `Function Defaults`, the `FunctionCustomizationTemplate` property is available for use with this method. When the `FunctionCustomizationTemplate` is set to `Default`, an additional property, `MemorySection`, becomes available for use with this method for these categories.

Data Defaults categories have the `StorageClass` property available for use with this method. The set of available properties is dynamic and varies based on the assigned value of the `StorageClass` property. For example, when the `StorageClass` property is assigned the value `ExportToFile`, which is allowed for certain categories, these additional properties become available for use with this method:

- "HeaderFile"
- "DefinitionFile"
- "Owner"
- "PreserveDimensions"

To learn more about storage classes and their properties, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

defaultPropVal — Property value

string scalar | character vector | true or 1 | false or 0

Default value to set for the specified property. The value type depends on the property whose value is being specified.

The available values for the `FunctionCustomizationTemplate`, `MemorySection`, and `StorageClass` are those defined in the dictionary, in the sections `FunctionCustomizationTemplates`, `MemorySections`, and `StorageClasses` respectively. To get the list of available values for each of these properties, use the `find` method in combination with the `getSection` method. For example, to get the list of available storage class names, enter:

```
find(getSection(coderDataDict,"StorageClasses")).Name
```

The available values for other properties depend on the type of property whose value is being specified. To learn more about storage class dependent properties and how to specify their values, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Examples

Set Default Entry for Receiver Interfaces

Create a Simulink data dictionary that contains an Embedded Coder Dictionary. Create the Embedded Coder Dictionary so that it represents a service interface configuration.

```
dictionaryFile = Simulink.data.dictionary.create("codeDefinitions.sldd");
codeDictionary = coder.dictionary.create("codeDefinitions.sldd","ServiceInterface");
```

Get the default receiver service interface.

```
receiver = getDictionaryDefault(codeDictionary,"DataReceiverInterfaces")
```

```
receiver =
```

Entry with properties:

```

                Name: 'ReceiverExample1'
        DataSource: 'C:\work\serviceDefinitions.sldd'
DataCommunicationMethod: 'OutsideExecution'
        FunctionName: 'get_$$N'
```

Get a list of the available receiver service interfaces.

```
recSection = getSection(codeDictionary, "DataReceiverInterfaces");  
receivers = find(recSection);  
receivers.Name
```

```
ans =
```

```
    'ReceiverExample1'
```

```
ans =
```

```
    'ReceiverExample2'
```

```
ans =
```

```
    'ReceiverExample3'
```

Set a different receiver interface as the default.

```
setDictionaryDefault(codeDictionary, "DataReceiverInterfaces", "ReceiverExample2");
```

Get and Set Default Outports Properties in Data Interface Coder Dictionary

Create a Simulink® data dictionary, then create the Embedded Coder Dictionary to represent a data interface configuration.

```
dictFileName = "dataDict.sldd";  
dataDict = Simulink.data.dictionary.create(dictFileName);  
coderDataDict = coder.dictionary.create(dataDict, "DataInterface");
```

Use the `getDictionaryDefault` method to see the default `StorageClass` of the `Outports` category.

```
getDictionaryDefault(coderDataDict, "Outports", "StorageClass")
```

```
ans =  
'Default'
```

Set the default `StorageClass` of the `Outports` category to `GetSet`. Then specify `GetFunction` (the naming rule of the get function) as `get_outports_$N` and specify `SetFunction` (the naming rule of the set function) as `set_outports_$N`.

```
setDictionaryDefault(coderDataDict, "Outports", "StorageClass", "GetSet")  
setDictionaryDefault(coderDataDict, "Outports", "GetFunction", "get_outports_$N")  
setDictionaryDefault(coderDataDict, "Outports", "SetFunction", "set_outports_$N")
```

Discard and close the dictionary, then delete the dictionary file. This allows you to run this example more than once without getting an error while attempting to create the dictionary.

```
discardChanges(dataDict)  
Simulink.data.dictionary.closeAll("-save")  
delete(dictFileName)
```

Version History

Introduced in R2023b

R2024a: Changes in service interfaces dictionary sections

Behavior changed in R2024a

The sections `SubcomponentInitTermFunctions` and `SubcomponentPeriodicAperiodicFunctions` replace the section `SubcomponentEntryFunctions`.

See Also

`coder.Dictionary` | `getDictionaryDefault`

unloadPackage

Class: `coder.Dictionary`

Namespace: `coder`

Unload package of code definitions from Embedded Coder Dictionary

Syntax

```
unloadPackage(coderDictionaryObj, pkgName)
```

Description

`unloadPackage(coderDictionaryObj, pkgName)` unloads the code definitions in the package `pkgName` from the Embedded Coder Dictionary that `coderDictionaryObj` represents.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

pkgName — package

character vector | string scalar

Package of code definitions to unload from the Embedded Coder Dictionary.

Example: 'Simulink'

Example: 'mpt'

Examples

Unload Package Definitions from Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Assign the `coder.Dictionary` object to the variable `coderDictObj`.

```
openExample('RollAxisAutopilot')  
coderDictObj = coder.dictionary.open('RollAxisAutopilot');
```

Unload the package `Simulink` from the dictionary. The definitions are not available for configuring model elements.

```
unloadPackage(coderDictObj, 'Simulink')
```

Version History

Introduced in R2019b

See Also

loadPackage

valid

Class: `coder.Dictionary`

Namespace: `coder`

Determine whether `coder.Dictionary` object represents a valid Embedded Coder Dictionary

Syntax

```
tf = valid(coderDictionaryObj)
```

Description

`tf = valid(coderDictionaryObj)` returns `true` if the `coder.Dictionary` object `coderDictionaryObj` is valid. A `coder.Dictionary` object is valid if it represents an existing Embedded Coder Dictionary. When you remove the Embedded Coder Dictionary, the `coder.Dictionary` object is not valid.

Input Arguments

coderDictionaryObj — Embedded Coder Dictionary object

`coder.Dictionary` object

Embedded Coder Dictionary, specified as a `coder.Dictionary` object.

Output Arguments

tf — True or false result

1 | 0 | logical array

True or false result, returned as a 1 or 0 of data type `logical`.

Examples

Check If Embedded Coder Dictionary Object Is Valid

Create a data dictionary named `'dataDict.sldd'` in the base workspace and create an Embedded Coder Dictionary in the data dictionary. Represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `myCoderDictObj` in the base workspace.

```
Simulink.data.dictionary.create('dataDict.sldd');  
myCoderDictObj = coder.dictionary.create('dataDict.sldd');
```

Check whether the `coder.Dictionary` object in the base workspace represents a valid Embedded Coder Dictionary.

```
valid(myCoderDictObj)
```

```
ans =
```

```
logical
```

```
1
```

Remove the Embedded Coder Dictionary from the data dictionary and check the `coder.Dictionary` object again. When you remove the Embedded Coder Dictionary, the `coder.Dictionary` object is not valid.

```
coder.dictionary.remove('dataDict.sldd');  
valid(myCoderDictObj)
```

```
ans =
```

```
logical
```

```
0
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary`

coder.dictionary.Entry Class

Namespace: `coder.dictionary`

Configure Embedded Coder Dictionary definition

Description

An object of the `coder.dictionary.Entry` class represents one definition of an Embedded Coder Dictionary. In this API, the object is called an *entry*. The information that the object represents is a *definition*. In this documentation, *definition* refers to the definition of the entry object.

A `coder.Dictionary` object contains `coder.dictionary.Section` objects, which represent the categories of an Embedded Coder Dictionary. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects, which represent the definitions stored in that section of the Embedded Coder Dictionary. The name of the section identifies the type of code definitions that the section contains. To access the sections of an Embedded Coder Dictionary, use a `coder.Dictionary` object. To access the dictionary entries within the section, use a `coder.dictionary.Section` object.

Creation

The functions `addEntry`, `getEntry`, `copyEntry`, and `find` create `coder.dictionary.Entry` objects.

Code Definition Properties

A `coder.dictionary.Entry` object has these properties.

Name — Name of code definition

character vector

Name of the coder dictionary definition that the entry represents. The name must be unique among the definitions in the section of the dictionary.

DataSource — Location of code definition

character vector

The location of the code definition. This property is read-only.

- **Built-in** — Provided by Simulink.
- **Model name** — Defined in a Simulink model.
- **Dictionary name** — Defined in a Simulink data dictionary.
- **Package name** — Defined in the Simulink package or in a custom package.

To access the properties of the code definition that an entry represents, use the `getAvailableProperties`, `get`, and `set` methods. Each type of code definition has available properties listed.

Storage Classes

Name — Name of storage class

StorageClass1 (default) | character vector | string scalar

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

character vector | string scalar

Custom text that you can use to describe the purpose and functionality of the storage class.

DataSource — Location of code definition

character vector

The location of the code definition. This property is read-only.

- Built-in — Provided by Simulink.
- Model name — Defined in a Simulink model.
- Dictionary name — Defined in a Simulink data dictionary.
- Package name — Defined in the Simulink package or in a custom package.

DataAccess — Specification to access the data

Direct (default) | Function

Specification to access data associated with the model. Access the data directly (**Direct**), through customizable **get** and **set** functions (**Function**), or by using a pointer (**Pointer**). For more information, see “Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary”.

Dependencies

- Setting this property to **Function** or **Pointer**:
 - Sets **DataScope** to **Imported**.
 - Means that you cannot specify multi-instance properties.
 - Sets the **PreserveDimensions** property to **false**. To preserve dimensions of multidimensional arrays in the generated code, set **DataAccess** to **Direct**.

In addition, setting this property to **Function** enables these properties:

- **AccessMode**
- **AllowedAccess**
- **GetFunctionName**
- **SetFunctionName**

DataScope — Specification to generate data definition

Exported (default) | Imported

Specification that the generated code defines 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 DefinitionFile. 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 HeaderFile 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 HeaderFile.

Header File — Name of header file that declares data

\$N.h (default) | character vector | string scalar

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 DataScope to Exported, you must use one of the tokens \$R or \$N in the value of this property.
- If you set DataScope 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) | character vector | string scalar

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
\$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 `DataScope` to `Imported` disables `DefinitionFile`. To include your external source code file in the build process, use model configuration parameters. For an example, see “Configure Data Interface”.

AccessMode — 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 `DataAccess` to `Function`.

AllowedAccess — Specification to allow access to data through functions

`ReadWrite` (default) | `ReadOnly` | `WriteOnly`

Specification for the storage class to allow read and write (`ReadWrite`), read-only (`ReadOnly`), or write-only (`WriteOnly`) access to the data.

Dependencies

This property is enabled only when you set `DataAccess` to `Function`.

GetFunctionName — Name of the get function that fetches the associated data

`get_ $N$M` (default) | character vector | string scalar

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
<code>\$N</code>	Name of associated data element (required)
<code>\$R</code>	Name of root model
<code>\$M</code>	Mangle text that ensures uniqueness
<code>\$U</code>	User token text. See “Identifier Format Control”.

Dependencies

This property is enabled only when you set `DataAccess` to `Function` and `AllowedAccess` to `ReadWrite` or `ReadOnly`.

SetFunctionName — Name of the set function that modifies the associated data

`set_ $N$M` (default) | character vector | string scalar

Name of the `set` function that fetches the modified 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
<code>\$N</code>	Name of associated data element (required)
<code>\$R</code>	Name of root model

Token	Description
\$M	Mangle text that ensures uniqueness
\$U	User token text. See “Identifier Format Control”.

Dependencies

This property is enabled only when you set `DataAccess` to `Function` and `AllowedAccess` to `ReadWrite` or `WriteOnly`.

DifferentInstanceDataSettings — Specification to assign separate storage settings for single-instance data and multi-instance data

`false` (default) | `true`

Specification for the storage class to use either the storage settings that you specify for single-instance data or the settings that you specify for multi-instance data. 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 properties `SingleInstanceStorageType`, `MultiInstanceStorageType`, and `MultiInstanceStructureTypeName`, `MultiInstanceStructureInstanceName`.

StorageType — 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 `StructureTypeName` and `StructureInstanceName`.

StructureTypeName — Name of structure type

`$R$N$G$M` (default) | character vector | string scalar

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 <code>step</code>
\$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 `StorageType` to `Structured` enables this property.

StructureInstanceName — Name of structure variable

\$G\$N\$M (default) | character vector | string scalar

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 <code>step</code>
\$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 `StorageType` to `Structured` enables this property.

StructureBitFields — Specification to represent structure members as bit fields

off (default) | on

Specification for the storage class to represent members of the structure as bit fields in the generated code.

This setting only applies to Boolean, fixed-point, and integer data types. Data types whose size exceed the size of an integer are not supported.

Storage classes with this property enabled cannot be assigned to data using the **Default Code Mappings** tab.

Dependencies

Setting `DataAccess` to `Direct`, `DataScope` to `Exported`, and `StorageType` to `Structured` enables this property.

SingleInstanceStorageType — Specification to aggregate single-instance data into a structure

Structured (default) | Unstructured

Specification to aggregate the single-instance 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 the property `UseDifferentPropSettingsForInstanceData` to `true` enables this property.
- Setting this property to `Structured` enables the properties `SingleInstanceStructTypeName` and `SingleInstanceStructureInstanceName`.

SingleInstanceStructTypeName — Name of structure type for single-instance data

\$R\$N\$G\$M (default) | character vector | string scalar

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 <code>step</code>
\$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 `SingleInstanceStorageType` to `Structured` enables this property.

SingleInstanceStructureInstanceName — Name of structure variable for single-instance data

`$G$N$M` (default) | character vector | string scalar

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 <code>step</code>
\$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 `SingleInstanceStorageType` to `Structured` enables this property.

SingleInstanceStructureBitFields — Specification to represent structure members for single-instance data as bit fields

`off` (default) | `on`

Specification for the storage class to represent members of the structure as bit fields in the generated code.

This setting only applies to Boolean, fixed-point, and integer data types. Data types whose size exceed the size of an integer are not supported.

Storage classes with this property enabled cannot be assigned to data using the **Default Code Mappings** tab.

Dependencies

Setting `DataAccess` to `Direct`, `DataScope` to `Exported`, and `SingleInstanceStorageType` to `Structured` enables this property.

MultiInstanceStorageType — Specification to aggregate multi-instance data into a structure

Structured (default)

Specification to aggregate the single-instance 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. You cannot change the value of this property.

Dependencies

Setting the property `UseDifferentPropSettingsForInstanceData` to `true` enables this property.

MultiInstanceStructureTypeName — Name of structure type for multi-instance data

`$R$N$G$M` (default) | character vector | string scalar

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
<code>\$R</code>	Name of root model
<code>\$N</code>	Base name of associated function, such as <code>step</code>
<code>\$G</code>	Name of storage class
<code>\$U</code>	User token text, which you specify for a model as described in “Identifier Format Control”
<code>\$M</code>	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

- Setting the property `UseDifferentPropSettingsForInstanceData` to `true` enables this property.
- If you set the property `DataAccess` to `Function`, you cannot set this property.

MultiInstanceStructureInstanceName — Name of structure variable for multi-instance data

`$G$N$M` (default) | character vector | string scalar

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
<code>\$R</code>	Name of root model
<code>\$N</code>	Base name of associated function, such as <code>step</code>
<code>\$G</code>	Name of storage class
<code>\$U</code>	User token text, which you specify for a model as described in “Identifier Format Control”
<code>\$M</code>	Name-mangling text inserted, if necessary, to avoid name collisions

Dependencies

- Setting the property `UseDifferentPropSettingsForInstanceData` to `true` enables this property.
- If you set the property `DataAccess` to `Function`, you cannot set this property.

MultiInstanceStructureBitFields — Specification to represent structure members for multi-instance data as bit fields`off (default) | on`

Specification for the storage class to represent members of the structure as bit fields in the generated code.

This setting only applies to Boolean, fixed-point, and integer data types. Data types whose size exceed the size of an integer are not supported.

Storage classes with this property enabled cannot be assigned to data using the **Default Code Mappings** tab.

Dependencies

Setting `DataAccess` to `Direct`, `DataScope` to `Exported`, and `MultiInstanceStorageType` to `Structured` enables this property.

DataInit — How to initialize data`Dynamic (default) | Static | None`

Specification that the generated codes 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.

For storage classes used for persistent data, you must set `DataInit` to `None`.

Dependencies

- If you set `Const` to `true`, you cannot set this property to `Dynamic`.
- Setting this property to `Dynamic` sets the property `Const` to `false`.

MemorySection — Location in memory to allocate data`None (default) | coder.dictionary.Entry object`

Location in memory to allocate data, specified as a `coder.dictionary.Entry` object that represents a memory section that exists in the Embedded Coder Dictionary. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

PreserveDimensions — Specification to preserve dimensions of multidimensional arrays`false (default) | true`

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

false (default) | true

Specification to apply the const qualifier to the data.

Dependencies

- If you select this property, you cannot set `DataInit` to `Dynamic`.
- Setting `DataInit` to `Dynamic` sets this property to false.

Volatile — Specification to apply volatile qualifier

false (default) | true

Specification to apply the volatile qualifier to the data.

OtherQualifier — Specification to apply a custom qualifier

character vector | string scalar

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”.

AccessibleByParameters — Whether to allow usage with model parameters

false (default) | true

Specification indicating whether to allow usage of the storage class with model parameters.

Dependencies

- Setting `DataInit` to `Static` sets this property to true.
- Setting `DataInit` to `Dynamic` sets this property to false.
- To set the value of this property, set `DataInit` to `None`.

AccessibleBySignals — Whether to allow usage with model signals

true (default) | false

Specification indicating whether to allow usage of the storage class with model signals.

Dependencies

- Setting `DataInit` to `Dynamic` sets this property to true.
- Setting `DataInit` to `Static` sets this property to false.
- To set the value of this property, set `DataInit` to `None`.

Function Customization Templates

Name — Name of function template

FunctionTemplate1 (default) | character vector | string scalar

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

character vector | string scalar

Custom text that you can use to describe the purpose and functionality of the function template.

FunctionName — Names of generated functions

\$R\$N (default) | character vector | string scalar

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 <code>step</code>
\$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

MemorySection — Location in memory to allocate function

None (default) | `coder.dictionary.Entry` object

Location in memory to allocate function, specified as a `coder.dictionary.Entry` object that represents a memory section that exists in the Embedded Coder Dictionary. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

Memory Sections

Name — Name of memory section

character vector | string scalar

Name of the memory section, specified as a character vector or string scalar. 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 <code>const</code> to the data.
MemVolatile	Apply the storage type qualifier <code>volatile</code> to the data.

Memory Section	Description
MemConstVolatile	Apply the storage type qualifiers <code>const</code> and <code>volatile</code> to the data.

Description — Purpose and functionality of memory section

character vector | string scalar

Custom text that you can use to describe the purpose and functionality of the memory section.

Comment — Comment to insert in the generated code

character vector | string scalar

Code comment that the code generator includes with the pragmas or other decorations that you specify with `PreStatement` and `PostStatement`.

PreStatement — Code to insert before data or function code

character vector | string scalar

Code, such as pragmas, to insert before the definitions and declarations of the data or functions that are in the memory section.

You can use the token `$R` to represent the name of the model that uses the memory section.

When you set `StatementsSurround` to `EachVariable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

PostStatement — Code to insert after data or function code

character vector | string scalar

Code, such as pragmas, to insert after the definitions and declarations of the data or functions that are in the memory section.

You can use the token `$R` to represent the name of the model that uses the memory section.

When you set `StatementsSurround` to `EachVariable`, you can use the token `$N` to represent the name of each variable or function that uses the memory section.

StatementsSurround — Specification to wrap data and functions separately or in a group`EachVariable` (default) | `AllVariables`

Specification to insert code statements (`PreStatement` and `PostStatement`):

- Around each variable and function that uses the memory section. Select `EachVariable`.
- 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 `AllVariables`.

Receiver Service Interfaces**Name — Name of receiver service**

character vector | string scalar

Name of the receiver service, specified as a character vector or string scalar. Embedded Coder provides the example services listed in this table.

Receiver Service	Description
ReceiverExample1	Favor memory optimization over data freshness. This service has <code>DataCommunicationMethod</code> set to <code>OutsideExecution</code> , so the service reads data from another function before the function starts executing.
ReceiverExample2	Favor data freshness over memory optimization. This service has <code>DataCommunicationMethod</code> set to <code>DuringExecution</code> , so the service reads data from another function immediately during execution.
ReceiverExample3	Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design. This service has <code>DataCommunicationMethod</code> set to <code>DirectAccess</code> , so the service reads data from generated function code directly by using memory that platform services manage for the target execution environment.

DataCommunicationMethod — How receiver data transmission is handled

`OutsideExecution` (default) | `DuringExecution` | `DirectAccess`

How receiver data transmission is handled between the target platform service and the run-time environment, specified as one of these options:

- `OutsideExecution` — Read data from another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- `DuringExecution` — Read data from another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.
- `DirectAccess` — Read data from generated function code directly by using memory that platform services manage. Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design.

FunctionName — Names of generated receiver functions

naming rule

Names of the receiver 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
\$G	Service name
\$N	Element name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in “Identifier Format Control”
\$M	Name-mangling text inserted to avoid name collisions

Dependencies

To specify a function naming rule, you must set `DataCommunicationMethod` to `OutsideExecution` or `DuringExecution`.

StorageClass — Storage class definition

character vector | string scalar

Storage class definition, specified as one of the definitions in the `StorageClasses` section of the dictionary.

Dependencies

To specify a storage class receiver services, you must set `DataCommunicationMethod` to `DirectAccess`.

Sender Service Interfaces**Name — Name of sender service**

character vector | string scalar

Name of the sender service, specified as a character vector or string scalar. Embedded Coder provides the example services listed in this table.

Sender Service	Description
SenderExample1	Favor memory optimization over data freshness. This service has <code>DataCommunicationMethod</code> set to <code>OutsideExecution</code> , so the service sends data to another function after the function executes.
SenderExample2	Favor data freshness over memory optimization. This service has <code>DataCommunicationMethod</code> set to <code>DuringExecution</code> , so the service reads data from another function immediately during execution.
SenderExample3	Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design. This service has <code>DataCommunicationMethod</code> set to <code>DirectAccess</code> , so the service reads data from generated function code directly by using memory that platform services manage for the target execution environment.

DataCommunicationMethod — How sender data transmission is handled`OutsideExecution` (default) | `DuringExecution` | `DirectAccess`

How sender data transmission is handled between the task and the run-time environment, specified as one of these options:

- **OutsideExecution** — Send data to another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- **DuringExecution** — Send data to another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.
- **DirectAccess** — Send data to generated function code directly by using memory that platform services manage. Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design.

FunctionNamingRuleForValue — Names of generated sender functions

naming rule

Names of the sender 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
\$G	Service name
\$N	Element name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in “Identifier Format Control”
\$M	Name-mangling text inserted to avoid name collisions

Dependencies

To specify a function naming rule, you must set `DataCommunicationMethod` to `OutsideExecution` or `DuringExecution`.

FunctionNamingRuleForValueReference — Names of generated sender functions that use reference naming rule

Names of the sender functions that send a value by reference 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
\$G	Service name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in “Identifier Format Control”

Dependencies

To specify a function naming rule, the definition must be a data sender service and you must set `DataCommunicationMethod` to `OutsideExecution` or `DuringExecution`.

StorageClass — Storage class definition

character vector | string scalar

Storage class definition, specified as one of the definitions in the `StorageClasses` section of the dictionary.

Dependencies

To specify a storage class receiver services, you must set `DataCommunicationMethod` to `DirectAccess`.

Data Transfer Service Interfaces

Name — Name of data transfer service

character vector | string scalar

Name of the data transfer service, specified as a character vector or string scalar. Embedded Coder provides the example services listed in this table.

Data Transfer Service	Description
DataTransferExample1	Favor memory optimization over data freshness. This service has <code>DataCommunicationMethod</code> set to <code>OutsideExecution</code> , so the service receives and sends data from another function outside of function execution.
DataTransferExample2	Favor data freshness over memory optimization. This service has <code>DataCommunicationMethod</code> set to <code>DuringExecution</code> , so the service reads data from another function immediately during execution.

DataCommunicationMethod — How data transmission is handled

`OutsideExecution` (default) | `DuringExecution`

How data transmission is handled between the task and the run-time environment, specified as one of these options:

- `OutsideExecution` — Communicate data to another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- `DuringExecution` — Communicate data to another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.

DataReceiverFunctionName — Names of generated receiver functions

naming rule

Names of the receiver 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	Element name
\$X	Current callable function
\$M	Name-mangling text inserted to avoid name collisions

DataSenderFunctionName — Names of generated sender functions

naming rule

Names of the sender 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	Element name
\$X	Current callable function
\$M	Name-mangling text inserted to avoid name collisions

Timer Services

Name — Name of timer service

character vector | string scalar

Name of the timer service, specified as a character vector or string scalar. Embedded Coder provides the example service listed in this table.

Timer Service	Description
TimerServiceExample1	The timer service executes outside of function execution.

DataCommunicationMethod — How data transmission is handled

OutsideExecution (default) | DuringExecution

How data transmission is handled between the task and the run-time environment, specified as one of these options:

- **OutsideExecution** — Communicate data to another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- **DuringExecution** — Communicate data to another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.

FunctionClockTickFunctionName — Clock tick function naming rule

naming rule

Names of the clock tick 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
\$G	Service name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in "Identifier Format Control"

Dependencies

- If you create multiple timer services, use the \$G token.
- If you set DataCommunicationMethod to OutsideExecution, use the \$X token.

Parameter Tuning Interfaces

Name — Name of parameter tuning service

"Not tunable" (default) | character vector | string scalar

The name of the parameter tuning service to use with parameters, specified as a character vector or string scalar. If the parameters do not need to be tuned, this property has the default value, "Not tunable".

StorageClass — Storage class definition

character vector | string scalar

Storage class definition, specified as one of the definitions in the `StorageClasses` section of the dictionary.

Parameter Argument Tuning Interfaces

Name — Name of parameter argument tuning service

"Not tunable" (default) | character vector | string scalar

The name of the parameter argument tuning service to use with parameter arguments, specified as a character vector or string scalar. If the parameter arguments do not need to be tuned, this property has the default value, "Not tunable".

StorageClass — Storage class definition

character vector | string scalar

Storage class definition, specified as one of the definitions in the `StorageClasses` section of the dictionary.

Measurement Service Interfaces

Name — Name of measurement service

"Not measured" (default) | character vector | string scalar

The name of the measurement service to use with signals, states, and internal data elements, specified as a character vector or string scalar. If the signals, states, and internal data elements do not need to be measured, this property has the default value, "Not measured".

IsPersistent — Option to use persistent data

false (default) | true

Option to store internal states and data store data in nonvolatile memory in the target environment. Specified as a logical false or true.

When `IsPersistent` is true, only storage classes with `DataInit` set to `None` are supported.

StorageClass — Storage class definition

character vector | string scalar

Storage class definition, specified as one of the definitions in the `StorageClasses` section of the dictionary.

Only storage classes with `DataInit` set to `None` are available for service interfaces that specify persistent data.

Methods

Public Methods

<code>get</code>	Get value of code definition property
<code>set</code>	Set value of code definition property
<code>getAvailableProperties</code>	Return properties for code definition
<code>isDictionaryDefault</code>	Determine if entry is default code definition for a category
<code>deleteEntry</code>	Delete Embedded Coder Dictionary entry
<code>valid</code>	Determine if <code>coder.dictionary.Entry</code> object represents a valid code definition

Examples

Create Definition in Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `coderDictionary`. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
openExample("RollAxisAutopilot")
coderDictionary = coder.dictionary.open("RollAxisAutopilot");
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, "StorageClasses");
```

Add a definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings. You can change these settings by using the `set` function.

```
newEntry = addEntry(storageClassesSect, "MyStorageClass")
```

```
newEntry =
```

```
    Entry with properties:
```

```
        Name: 'MyStorageClass'
    DataSource: 'RollAxisAutopilot'
```

Version History

Introduced in R2019b

R2025a: Measurement service interface for persistent data support for target platform nonvolatile memory

Specify a measurement service interface for persistent data to support state and data store data that is stored in nonvolatile memory in the target environment.

To specify persistent data, use the service interface `PermanentRAM` in the `MeasurementInterfaces` section of the coder dictionary. Alternatively, you can define a new service interface in the `MeasurementInterfaces` section, where you set property `IsPersistent` to `true` for the service and you specify a storage class that has `DataInit` set to `None`. A new storage class called `PersistentMemory` is provided in the `StorageClasses` section of the coder dictionary.

R2024a: Not measured and Not tunable as dictionary default

You can now specify "Not measured" as the default value of the `MeasurementInterfaces` section in the coder dictionary, and you can specify "Not tunable" as the default value of the `ParameterTuningInterfaces` and `ParameterArgumentTuningInterfaces` sections in the dictionary. The "Not measured" and "Not tunable" entries are included in the dictionary, and are preselected as default values in newly created dictionaries. You can select other entries as dictionary default for these sections, but you cannot delete these entries from the dictionary.

R2023b: Support for service interface configurations

Access the `coder.dictionary.Entry` objects for Embedded Coder Dictionaries that use the service interface configuration. You can still access the entry objects for Embedded Coder Dictionaries that use the data interface configuration.

See Also

`coder.Dictionary` | `coder.dictionary.Section` | Embedded Coder Dictionary

deleteEntry

Class: `coder.dictionary.Entry`

Namespace: `coder.dictionary`

Delete Embedded Coder Dictionary entry

Syntax

```
deleteEntry(entryObj)
```

Description

`deleteEntry(entryObj)` deletes an Embedded Coder Dictionary definition that the entry `entryObj` represents.

Input Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, specified as a `coder.dictionary.Entry` object.

Examples

Delete Storage Class from Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect, 'ParamStruct');
```

Delete the `ParamStruct` storage class.

```
deleteEntry(entryObj)
```

Version History

Introduced in R2019b

See Also

[coder.dictionary.Entry](#) | [coder.Dictionary](#) | [coder.dictionary.Section](#)

get

Class: `coder.dictionary.Entry`

Namespace: `coder.dictionary`

Get value of code definition property

Syntax

```
propValue = get(entryObj,propName)
```

Description

`propValue = get(entryObj,propName)` returns the value of the property `propName` for the code definition that `entryObj` represents.

Input Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

Embedded Coder Dictionary entry that represents the code definition, specified as a `coder.dictionary.Entry` object. Before you use this function, represent the code definition by using a `coder.dictionary.Entry` object. Use, for example, the `addEntry` function.

propName — property name

character vector | string scalar

Property of code definition, specified as a character vector or string scalar.

Output Arguments

propValue — property value

string | `coder.dictionary.Entry` object

Value of the property `propName` for the code definition that `entryObj` represents, returned as a string or a `coder.dictionary.Entry` object. For storage classes and function customization templates, the `MemorySection` property can have a value that is a `coder.dictionary.Entry` object representing a memory section definition.

Examples

Return Property Value of Storage Class Definition

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section with a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class ParamStruct by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect, 'ParamStruct');
```

Return the value of the `DataInit` property of the storage class definition. This storage class uses static data initialization.

```
dataInitVal = get(entryObj, 'DataInit')
```

```
dataInitVal =
```

```
    'Static'
```

Version History

Introduced in R2019b

See Also

`coder.dictionary.Entry` | `coder.Dictionary` | `coder.dictionary.Section`

getAvailableProperties

Class: `coder.dictionary.Entry`

Namespace: `coder.dictionary`

Return properties for code definition

Syntax

```
properties = getAvailableProperties(entryObj)
```

Description

`properties = getAvailableProperties(entryObj)` returns a cell array of the properties of the code definition that `entryObj` represents.

Input Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, specified as a `coder.dictionary.Entry` object.

Output Arguments

properties — code definition properties

cell array of property names

Code definition properties available for the definition that `entryObj` represents, returned as a cell array of character vectors. For a list of properties available for each type of code definition, see Embedded Coder Dictionary.

Examples

Get Properties of Storage Class Definition

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect, 'ParamStruct');
```

Get a list of the properties of the `ParamStruct` storage class. To get and set the values of these properties, use the `get` and `set` methods.

```
properties = getAvailableProperties(entryObj)
```

Version History

Introduced in R2019b

See Also

`coder.dictionary.Entry` | `coder.Dictionary` | `coder.dictionary.Section`

isDictionaryDefault

Class: `coder.dictionary.Entry`

Namespace: `coder.dictionary`

Determine if entry is default code definition for a category

Syntax

```
tf = isDictionaryDefault(entryObj)
```

Description

`tf = isDictionaryDefault(entryObj)` returns `true` if the entry is the default code definition for its category.

Input Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, specified as a `coder.dictionary.Entry` object.

Output Arguments

tf — True or false result

1 | 0 | logical array

True or false result, returned as a 1 or 0 of data type `logical`.

Examples

Check if Entry Object Is Default Measurement Interface

Create a Simulink data dictionary that contains an Embedded Coder Dictionary. Create the Embedded Coder Dictionary so that it represents a service interface configuration.

```
dictionaryFile = Simulink.data.dictionary.create('codeDefinitions.sldd');  
codeDictionary = coder.dictionary.create('codeDefinitions.sldd','ServiceInterface');
```

Create a `coder.dictionary.Section` object that represents the **Measurement Interfaces** section of the Embedded Coder Dictionary.

```
measSect = getSection(codeDictionary, 'MeasurementInterfaces');
```

Get the `coder.dictionary.Entry` objects that represent the definitions in the **Measurement Interfaces** section. For this example, there is one measurement interface entry.

```
entries = find(measSect)
```

```
entries =
```

Entry with properties:

```
        Name: 'MeasurementExample1'  
    DataSource: 'C:\work\codeDefinitions.sldd'  
    StorageClass: 'MeasurementStruct'
```

Determine if the entry `MeasurementExample1` is the default measurement interface for the dictionary.

```
tf = isDictionaryDefault(entries)
```

```
tf =
```

```
    logical
```

```
    1
```

For this example, the measurement interface entry is the default code definition for measurement interfaces.

Version History

Introduced in R2023b

See Also

`coder.dictionary.Entry` | `coder.Dictionary` | `coder.dictionary.Section`

set

Class: `coder.dictionary.Entry`

Namespace: `coder.dictionary`

Set value of code definition property

Syntax

```
set(entryObj,Name1,Value1,...,NameN,ValueN)
```

Description

`set(entryObj,Name1,Value1,...,NameN,ValueN)` sets the properties to the specified values for the Embedded Coder Dictionary definition that `entryObj` represents.

Input Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

Embedded Coder Dictionary entry that represents the code definition, specified as a `coder.dictionary.Entry` object. Before you use this function, represent the code definition by using a `coder.dictionary.Entry` object. Use, for example, the `addEntry` function.

Name1,Value1,...,NameN,ValueN — Property specifications

name-value pairs representing properties and values

Property specifications for the definition, specified as one or more name-value pairs representing names and values of properties of the code definition. For a list of the properties of an Embedded Coder Dictionary definition, see `coder.dictionary.Entry`.

Example: `'MemorySection','memSect1'`

Example: `'DataScope','Exported'`

Examples

Set Property Value of Storage Class Definition

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect,'ParamStruct');
```


Change the value of the `HeaderFile` property of the storage class definition from `$N.h` to `$N_params.h`.

```
set(entryObj, 'HeaderFile', '$N_params.h')
```

Version History

Introduced in R2019b

See Also

`coder.dictionary.Entry` | `coder.Dictionary` | `coder.dictionary.Section`

valid

Class: `coder.dictionary.Entry`

Namespace: `coder.dictionary`

Determine if `coder.dictionary.Entry` object represents a valid code definition

Syntax

```
tf = valid(entryObj)
```

Description

`tf = valid(entryObj)` returns `true` if the `coder.dictionary.Entry` object `entryObj` is valid. A `coder.dictionary.Entry` object is valid if it represents an existing code definition. When you remove the code definition that the entry represents, the `coder.dictionary.Entry` object is not valid.

Input Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, specified as a `coder.dictionary.Entry` object.

Output Arguments

tf — True or false result

1 | 0 | logical array

True or false result, returned as a 1 or 0 of data type `logical`.

Examples

Check if Entry Object Is Valid

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary and represent the section by using a `coder.dictionary.Section` object.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect, 'ParamStruct');
```

Check if the `coder.dictionary.Entry` object in the base workspace represents a valid code definition.

```
valid(entryObj)
```

```
ans =
```

```
    logical
```

```
    1
```

Remove the code definition from the Embedded Coder Dictionary and check the `coder.dictionary.Entry` object again. When you remove the code definition, the `coder.dictionary.Entry` object is not valid.

```
deleteEntry(storageClassesSect, 'ParamStruct');  
valid(entryObj)
```

```
ans =
```

```
    logical
```

```
    0
```

Version History

Introduced in R2019b

See Also

`coder.dictionary.Entry` | `coder.Dictionary` | `coder.dictionary.Section`

addEntry

Class: `coder.dictionary.Section`

Namespace: `coder.dictionary`

Add a new entry to Embedded Coder Dictionary section

Syntax

```
addEntry(sectionObj,defName)
entryObj = addEntry(sectionObj,entryName)
```

Description

`addEntry(sectionObj,defName)` adds a definition named `defName` to the Embedded Coder Dictionary section `sectionObj`, a `coder.dictionary.Section` object.

`entryObj = addEntry(sectionObj,entryName)` returns a `coder.dictionary.Entry` object that represents the new Embedded Coder Dictionary definition.

Input Arguments

sectionObj — Embedded Coder Dictionary section

`coder.dictionary.Section` object

Section in the Embedded Coder Dictionary, specified as a `coder.dictionary.Section` object. The section name identifies the type of code definition that `addEntry` creates.

defName — Name of Embedded Coder Dictionary definition

character vector | string scalar

Name of newly created Embedded Coder Dictionary definition, specified as a character vector or string scalar.

Example: `'StorageClass2'`

Output Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

New Embedded Coder Dictionary entry, returned as a `coder.dictionary.Entry` object. The new entry represents the new code definition in the section of the Embedded Coder Dictionary.

Examples

Add Storage Class to Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
openExample('RollAxisAutopilot')
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Add a storage class definition named `MyStorageClass` to the Storage Classes section. The storage class definition uses the default property settings. You can change these settings by using the `set` function.

```
newEntry = addEntry(storageClassesSect, 'MyStorageClass')
```

```
newEntry =
```

```
    Entry with properties:
```

```
        Name: 'MyStorageClass'
    DataSource: 'RollAxisAutopilot'
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary` | `coder.dictionary.Section` | `coder.dictionary.Entry`

deleteEntry

Class: `coder.dictionary.Section`

Namespace: `coder.dictionary`

Delete Embedded Coder Dictionary entry

Syntax

```
deleteEntry(sectionObj,defName)
```

Description

`deleteEntry(sectionObj,defName)` deletes an Embedded Coder Dictionary definition that has the name `defName` from the section `sectionObj`, a `coder.dictionary.Section` object.

Input Arguments

sectionObj — Embedded Coder Dictionary section

`coder.dictionary.Section` object

Section in the Embedded Coder Dictionary that contains its definition, specified as a `coder.dictionary.Section` object.

defName — Name of Embedded Coder Dictionary definition

character vector | string scalar

Name of Embedded Coder Dictionary definition, specified as a character vector or string scalar. To get a list of definitions in a section, use the methods `find` and `get`.

Example: `'StorageClass2'`

Examples

Delete Storage Class from Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
openExample('RollAxisAutopilot')  
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses')
```

```
storageClassesSect =
```

```
    Section with properties:
```

```
Name: 'StorageClasses'
```

Get a list of the storage class definitions in the section. Use the `find` method to get the `coder.dictionary.Entry` objects in the section. Then, use the `get` method to list the names of the definitions that the entries represent.

```
entryObjects = find(coderDictionary, 'StorageClasses');
get(entryObjects, 'Name')
```

```
ans =
```

```
1×19 cell array
```

```
Columns 1 through 4
```

```
{'ExportedGlobal'} {'ImportedExtern'} {'ImportedExternP...'} {'BitField'}
```

```
Columns 5 through 9
```

```
{'Const'} {'Volatile'} {'ConstVolatile'} {'Define'} {'ImportedDefine'}
```

```
Columns 10 through 13
```

```
{'ExportToFile'} {'ImportFromFile'} {'FileScope'} {'Localizable'}
```

```
Columns 14 through 18
```

```
{'Struct'} {'GetSet'} {'CompilerFlag'} {'Reusable'} {'SignalStruct'}
```

```
Column 19
```

```
{'ParamStruct'}
```

Remove the example storage class `ParamStruct` from the dictionary.

```
deleteEntry(storageClassesSect, 'ParamStruct');
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary` | `coder.dictionary.Section` | `coder.dictionary.Entry`

exist

Class: `coder.dictionary.Section`

Namespace: `coder.dictionary`

Determine if code definition exists in Embedded Coder Dictionary section

Syntax

```
tf = exist(sectionObj,defName)
```

Description

`tf = exist(sectionObj,defName)` returns `true` if the Embedded Coder Dictionary section represented by `sectionObj` contains a definition with the name `defName`.

Input Arguments

sectionObj — Embedded Coder Dictionary section

`coder.dictionary.Section` object

Section in the Embedded Coder Dictionary, specified as a `coder.dictionary.Section` object.

defName — Name of Embedded Coder Dictionary definition

character vector | string scalar

Name of Embedded Coder Dictionary definition, specified as a character vector or string scalar.

Example: `'StorageClass2'`

Output Arguments

tf — True or false result

1 | 0 | logical array

True or false result, returned as a 1 or 0 of data type `logical`.

Examples

Check If Storage Class Exists

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
openExample('RollAxisAutopilot')  
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.


```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Check if the Storage Classes section contains the example storage class `SignalStruct`.

```
exist(storageClassesSect, 'SignalStruct')
```

```
ans =
```

```
logical
```

```
1
```

Remove the storage class, and then see whether the definition still exists.

```
deleteEntry(storageClassesSect, 'SignalStruct');
```

```
exist(storageClassesSect, 'SignalStruct')
```

```
ans =
```

```
logical
```

```
0
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary` | `coder.dictionary.Section` | `coder.dictionary.Entry`

find

Class: `coder.dictionary.Section`

Namespace: `coder.dictionary`

Search in Embedded Coder Dictionary section

Syntax

```
foundEntries = find(sectionObj,Name1,Value1,...,NameN,ValueN)
```

Description

`foundEntries = find(sectionObj,Name1,Value1,...,NameN,ValueN)` searches the Embedded Coder Dictionary section `sectionObj` by using search criteria `Name1,Value1,...,NameN,ValueN`, and returns an array of matching entries that were found in that section.

Input Arguments

sectionObj — Embedded Coder Dictionary section

`coder.dictionary.Section` object

Section to search in the Embedded Coder Dictionary, specified as a `coder.dictionary.Section` object.

Name1,Value1,...,NameN,ValueN — Search criteria

name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the definitions in the Embedded Coder Dictionary section. For a list of the properties of an Embedded Coder Dictionary entry, see `coder.dictionary.Entry`.

Example: 'MemorySection','memSect1'

Example: 'DataScope','Exported'

Output Arguments

foundEntries — Embedded Coder Dictionary entries matching search criteria

array of `coder.dictionary.Entry` objects

Embedded Coder Dictionary entries that match the specified search criteria, returned as an array of `coder.dictionary.Entry` objects.

Examples

Search Embedded Coder Dictionary for Exported Storage Classes

Open the model `RollAxisAutopilot` and create an Embedded Coder Dictionary in the model. Represent the Embedded Coder Dictionary by using the `coder.Dictionary` object

`coderDictionary`. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
openExample('RollAxisAutopilot')  
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Find the storage classes that have `DataScope` set to `Exported`.

```
exportedSCs = find(storageClassesSect, 'DataScope', 'Exported')  
exportedSCs =
```

```
1×2 Entry array with properties:
```

```
    Name  
DataSource
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary` | `coder.dictionary.Section` | `coder.dictionary.Entry`

getEntry

Class: `coder.dictionary.Section`

Namespace: `coder.dictionary`

Return `coder.dictionary.Entry` object that represents an Embedded Coder Dictionary definition

Syntax

```
entryObj = getEntry(sectionObj,defName)
```

Description

`entryObj = getEntry(sectionObj,defName)` returns a `coder.dictionary.Entry` object representing the Embedded Coder Dictionary definition that has the name `defName` found in the section `sectionObj`, a `coder.dictionary.Section` object.

Input Arguments

sectionObj — Embedded Coder Dictionary section

`coder.dictionary.Section` object

Embedded Coder Dictionary section containing the code definition, specified as a `coder.dictionary.Section` object. The section name identifies the type of code definition.

defName — Name of Embedded Coder Dictionary definition

character vector | string scalar

Name of Embedded Coder Dictionary definition, specified as a character vector or string scalar.

Example: 'StorageClass2'

Output Arguments

entryObj — Embedded Coder Dictionary entry

`coder.dictionary.Entry` object

Embedded Coder Dictionary entry, returned as a `coder.dictionary.Entry` object. The entry represents the code definition in the Embedded Coder Dictionary section.

Examples

Get Storage Class from Embedded Coder Dictionary

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using a `coder.Dictionary` object. Use this object to access the Storage Classes section of the dictionary, which contains the storage class definitions.

```
openExample('RollAxisAutopilot')  
coderDictionary = coder.dictionary.open('RollAxisAutopilot');
```

Create a `coder.dictionary.Section` object that represents the Storage Classes section of the Embedded Coder Dictionary.

```
storageClassesSect = getSection(coderDictionary, 'StorageClasses');
```

Represent the example storage class `ParamStruct` by using a `coder.dictionary.Entry` object.

```
entryObj = getEntry(storageClassesSect, 'ParamStruct')
```

Entry with properties:

```
    Name: 'ParamStruct'  
DataSource: 'RollAxisAutopilot'
```

Version History

Introduced in R2019b

See Also

`coder.Dictionary` | `coder.dictionary.Section` | `coder.dictionary.Entry`

coder.dictionary.Section Class

Namespace: `coder.dictionary`

Configure Embedded Coder Dictionary section

Description

An object of the `coder.dictionary.Section` class represents one section of an Embedded Coder Dictionary, such as Storage Classes or Data Sender Services. The object enables you to perform operations on the section such as adding or deleting entries.

A `coder.Dictionary` object contains `coder.dictionary.Section` objects, which represent the categories of an Embedded Coder Dictionary. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects, which represent the definitions stored in that section of the Embedded Coder Dictionary. The name of the section identifies the type of code definitions that the section contains. To access the sections of an Embedded Coder Dictionary, use a `coder.Dictionary` object. To access the dictionary entries within the section, use a `coder.dictionary.Section` object.

Creation

The function `getSection` creates a `coder.dictionary.Section` object.

Properties

Name — Name of Embedded Coder Dictionary Section

`string scalar` | `character vector`

This property is read-only.

Name of Embedded Coder Dictionary Section, stored as a character vector. This property is read-only. The sections in the dictionary depend on the code interface configuration type of the dictionary.

A service interface configuration contains these sections:

- `'InitTermFunctions'`
- `'PeriodicAperiodicFunctions'`
- `'DataReceiverInterfaces'`
- `'DataSenderInterfaces'`
- `'DataTransferInterfaces'`
- `'TimerInterfaces'`
- `'ParameterTuningInterfaces'`
- `'ParameterArgumentTuningInterfaces'`
- `'MeasurementInterfaces'`
- `'SubcomponentInitTermFunctions'`

- 'SubcomponentPeriodicAperiodicFunctions'
- 'SharedUtilityFunctions'
- 'InternalData'
- 'Constants'
- 'StorageClasses'
- 'DataMemorySections'
- 'FunctionMemorySections'

A data interface configuration contains these sections:

- 'StorageClasses'
- 'MemorySections'
- 'FunctionCustomizationTemplates'

Methods

Public Methods

addEntry	Add a new entry to Embedded Coder Dictionary section
getEntry	Return coder.dictionary.Entry object that represents an Embedded Coder Dictionary definition
copyEntry	Copy Embedded Coder Dictionary entry
deleteEntry	Delete Embedded Coder Dictionary entry
exist	Determine if code definition exists in Embedded Coder Dictionary section
find	Search in Embedded Coder Dictionary section

Examples

Create Embedded Coder Dictionary Section Object

Open the model `RollAxisAutopilot` and represent the Embedded Coder Dictionary by using the `coder.Dictionary` object `coderDictionary`.

```
openExample("RollAxisAutopilot")
coderDictionary = coder.dictionary.open("RollAxisAutopilot");
```

Create a `coder.dictionary.Section` object that represents the Memory Sections section of the Embedded Coder Dictionary.

```
memorySectionsSect = getSection(coderDictionary, "MemorySections")
```

```
memorySectionsSect =
```

```
    Section with properties:
```

```
        Name: 'MemorySections'
```

The section contains `coder.dictionary.Entry` objects, each of which represents a built-in memory section definition.

Version History

Introduced in R2019b

R2024a: Changes in service interfaces dictionary sections

- Constants and InternalData are new dictionary sections.
- The dictionary sections SubcomponentInitTermFunctions and SubcomponentPeriodicAperiodicFunctions replace the section SubcomponentEntryFunctions.

R2023b: Support for service interface configurations

Access the sections of Embedded Coder Dictionaries that use the service interface configuration. You can still access sections in Embedded Coder Dictionaries that use the data interface configuration.

See Also

`coder.Dictionary` | `coder.dictionary.Entry` | Embedded Coder Dictionary

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 this 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 might 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 C Code

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

Write MATLAB Function

Write a MATLAB function that you want to replace with a custom implementation, `replacement_calculate_impl.c`. In the MATLAB function, call `coder.replace`. For example, use the function `calculate`.

```
type calculate.m
```

```
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`.

type `top_function.m`

```
function out = top_function(in)  
arguments  
    in (1,1) double  
end  
    p = calculate(in);  
    out = exp(p);  
end
```

Create and Register Code Replacement Library

The code replacement library file `crl_matlab_fcn_coder_replace.m` 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.

type `crl_matlab_fcn_coder_replace.m`

```
function hTable = crl_matlab_fcn_coder_replace  
% Create a function to call the code replacement library table  
  
%% Create a table object  
hTable = RTW.TflTable;  
  
%% Create an entry  
hEntry = RTW.TflCFunctionEntry;  
  
%% Create entry parameters  
setTflCFunctionEntryParameters(hEntry, ...  
    'Key','calculate', ...  
    'Priority',100, ...  
    'ArrayLayout','COLUMN_MAJOR', ...  
    'ImplementationName', ...  
    'replacement_calculate_impl', ...  
    'ImplementationHeaderFile', ...  
    'replacement_calculate_impl.h', ...  
    'ImplementationSourceFile', ...  
    'replacement_calculate_impl.c');  
  
%% Create the conceptual representation  
arg = getTflArgFromString(hEntry,'y1','double');  
arg.IOType = 'RTW_IO_OUTPUT';  
addConceptualArg(hEntry,arg);  
  
arg = getTflArgFromString(hEntry,'u1','double');  
addConceptualArg(hEntry,arg);  
  
%% Create the implementation representation  
arg = getTflArgFromString(hEntry,'y1','double');  
arg.IOType = 'RTW_IO_OUTPUT';  
hEntry.Implementation.setReturn(arg);
```

```
arg = getTflArgFromString(hEntry,'u1','double');
hEntry.Implementation.AddArgument(arg);
```

```
%% Add the entry to the table
hTable.addEntry(hEntry);
```

An `rtwTargetInfo` file registers the code replacement table. For example, use the content from the file `RTWTargetInfo-coder-replace.txt`. To refresh the registration information, call `sl_refresh_customizations`.

```
copyfile RTWTargetInfo-coder-replace.txt rtwTargetInfo.m
sl_refresh_customizations
```

Apply Code Replacement Library and Generate Code

Create a library code generation configuration and use it to specify the code replacement library that you created.

```
configObj = coder.config('lib');
configObj.CodeReplacementLibrary = "CRL for matlab function with custom code using coder.replace";
```

Generate the code by using the code generation configuration.

```
codegen top_function.m -config configObj -c
```

Code generation successful.

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 code in which the replacement function `replacement_calculate_impl` replaces the MATLAB function `calculate`.

```
file = fullfile("codegen/lib/top_function","top_function.c");
coder.example.extractLines(file,"double top_function(double in)","",1,1);
```

```
double top_function(double in)
{
    return exp(replacement_calculate_impl(in));
}
```

Input Arguments

ifNoReplacement — Option to produce a warning or error when no match is found

`'-errorifnoreplacement' | '-warnifnoreplacement'`

Option to produce an error or warning when no match is found, specified as `'-errorifnoreplacement'` or `'-warnifnoreplacement'`. If you do not specify this argument, 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('-errorifnoreplacement')`

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.

Version History

Introduced in R2012b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`codegen`

Topics

“Replace MATLAB Functions with Custom Code by Using the `coder.replace` Function”

“Define Code Replacement Library Optimizations”

“What Is Code Replacement Customization?”

“What Is Code Replacement?”

coder.report.generateCodeMetrics

Generate static code metrics report

Syntax

```
coder.report.generateCodeMetrics(model)
coder.report.generateCodeMetrics(subsystem)
coder.report.generateCodeMetrics( ___,Name,Value)
```

Description

`coder.report.generateCodeMetrics(model)` generates a static code metrics report for the code generated from `model` without generating a full code generation report. The default file name of the static code metrics report is `metrics.html`.

You can also generate a static code metrics report in a code generation report by using the function `coder.report.generate` or by selecting **Generate Static Code Metrics for Model** in the Configuration Parameters dialog box.

`coder.report.generateCodeMetrics(subsystem)` generates the static code metrics report for the subsystem. The build folder for the subsystem must be present in the current working folder.

`coder.report.generateCodeMetrics( ___,Name,Value)` specifies options by using one or more `Name,Value` pair arguments.

Examples

Generate Static Code Metrics for Model

Generate code for a model, and then generate the static code metrics.

Open the model `CounterModel`.

```
openExample('CounterModel.slx')
```

Turn off code generation report generation.

```
set_param('CounterModel','GenerateReport','off');
```

Build the model.

```
slbuild('CounterModel');
```

Generate a static code metrics report for the model.

```
coder.report.generateCodeMetrics('CounterModel');
```

Open the static code metrics report.

```
web('metrics.html');
```

Generate Static Code Metrics for Subsystem

Generate code and static code metrics for a subsystem.

Open the model CounterModel.

```
openExample('CounterModel.slx')
```

Turn off code generation report generation.

```
set_param('CounterModel','GenerateReport','off');
```

Build the subsystem.

```
slbuild('CounterModel/Amplifier');
```

Generate a static code metrics report for the subsystem.

```
coder.report.generateCodeMetrics('CounterModel/Amplifier');
```

Open the static code metrics report.

```
web('metrics.html');
```

Specify Code Metrics Report File Name

Generate code metrics in a report that uses a custom file name.

Open the model CounterModel.

```
openExample('CounterModel.slx')
```

Turn off code generation report generation.

```
set_param('CounterModel','GenerateReport','off');
```

Build the model.

```
slbuild('CounterModel');
```

Generate a static code metrics report for the model. Name the generated code metrics report code_metrics.html.

```
coder.report.generateCodeMetrics('CounterModel','FileName','code_metrics.html');
```

Open the static code metrics report.

```
web('code_metrics.html');
```

Input Arguments

model — Model name

character vector | string scalar

Model name, specified as a character vector or string scalar.

Example: 'CounterModel'

Data Types: char

subsystem — Subsystem name

character vector | string scalar

Subsystem name, specified as a character vector or sting scalar.

Example: 'CounterModel/Amplifier'

Data Types: char

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: 'FileName', 'X:\code_metrics.html' creates a static code metrics report named `code_metrics.html`.

BuildDir — Build folder

character vector | string scalar

Build folder that contains the generated code, specified as a character vector or string scalar.

Example: 'BuildDir', 'X:\CustomCodeComments_ert_rtw'

FileName — Name of report file

character vector | string scalar

Name of the generated static code metrics HTML file, specified as a character vector or string scalar.

Example: 'FileName', 'X:\code_metrics.html'

Version History

Introduced in R2020b

See Also

`coder.report.generate`

Topics

“Static Code Metrics”

“Generate Static Code Metrics Report for Simulink Model”

`coder.setupMISRAConfig`

Configure parameters to improve generated code compliance with MISRA and AUTOSAR guidelines

Syntax

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

Description

`coder.setupMISRAConfig(cfg)` sets up an Embedded Coder code generation configuration object with properties selected to improve the generated code compliance with MISRA™ C:2012, MISRA C:2023, MISRA C++:2008, and AUTOSAR C++14 guidelines.

Examples

Configure Code Generation Configuration Object for Improved MISRA and AUTOSAR Compliance

Create an Embedded Coder code generation configuration object.

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

Set properties that might impact MISRA and AUTOSAR compliance.

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

The function `coder.setupMISRAConfig` sets property values according to the values shown in the table.

Property	Value for Improved MISRA Compliance
CastingMode	'Standards'
CppGenerateEnumClass (if target language is C++)	true
CppInterfaceClassName (if target language is C++)	' '
CppInterfaceStyle (if target language is C++)	'Functions'
CppNamespace (if target language is C++)	Valid namespace name for C++
DataTypeReplacement	'CoderTypedefs'
EnableDynamicMemoryAllocation	false
EnableOpenMP	false
EnableRuntimeRecursion	false
EnableSignedLeftShifts	false
EnableSignedRightShifts	false

Property	Value for Improved MISRA Compliance
GenerateDefaultInSwitch	true
HeaderGuardStyle	'UseIncludeGuard'
JustifyMISRAViolations	true
LargeConstantGeneration	'KeepInSourceFiles'
MaxIdLength (if target language is C)	31
ParenthesesLevel	'Maximum'
RuntimeChecks	false
TargetLangStandard	'C99 (ISO)' for C, 'C++11 (ISO)' for C++

Note GenerateComments must be set to true to add MISRA and AUTOSAR annotations by using JustifyMISRAViolations.

If the CppNamespace property is unset, and the TargetLang property is 'C++', then coder.setupMISRAConfig sets the CppNamespace property to a default character vector, 'Codegen'. Modify this value to a namespace name that is particular to your project.

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)`

Version History

Introduced in R2017b

See Also

coder.config | coder.EmbeddedCodeConfig | coder.CodeConfig

Topics

"Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance"

External Websites

www.misra.org.uk

www.autosar.org

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) %#codegen
% 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 Files */
#include "addglobals_ex_initialize.h"
#include "addglobals_ex.h"
#include "addglobals_ex_data.h"

/* Function Definitions */

/*
 * Arguments      : void
 * Return Type    : void
 */
void addglobals_ex_initialize(void)
{
    myglobalone = 1.0;
    isInitialized_addglobals_ex = true;
}
```

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_impnr`, assign the 'ImportedExternPointer' storage class to the global variable `myglobal`.

```
function y = addglobal_impnr(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 <i>entry_point_name.c</i>. - Declares the variable as an extern in the Variable Declarations section of the header file <i>entry_point_name.h</i> - Initializes the variable in the function <i>entry_point_name_initialize.h</i>.
'ExportedDefine'	Declares the variable with a <code>#define</code> directive in the Exported data define section of the header file <i>entry_point_name.h</i> .
'ImportedExtern'	Declares the variable as an extern in the Variable Declarations section of the header file <i>entry_point_name_data.h</i> . 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 <i>entry_point_name_data.h</i> . 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.

Version History

Introduced in R2015b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

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”

compare

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

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

“Manage Configuration Sets for a Model”

“Programmatic Code Generation Verification”

coder.MATLABCodeTemplate Class

Namespace: 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`.

The `coder.MATLABCodeTemplate` class is a handle class.

Creation

Description

`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 — Code generation template

character vector

Name of code generation template file, specified as a character vector.

Methods

Public Methods

<code>coder.MATLABCodeTemplate.emitSection</code>	Emit comments for template section
<code>coder.MATLABCodeTemplate.getcurrenttokens</code>	Get current tokens
<code>coder.MATLABCodeTemplate.getTokenValue</code>	Get value of token
<code>coder.MATLABCodeTemplate.setTokenValue</code>	Set value of token for code generation template

Examples

Create coder.MATLABCodeTemplate Object from Default CGT File

This example show how to create a `coder.MATLABCodeTemplate` object from the default code generation template (CGT) file *matlabroot*/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt

```
newObj = coder.MATLABCodeTemplate()

newObj =

    MATLABCodeTemplate with properties:

        CGTFile: 'matlabcoder_default_template.cgt'
```

Create coder.MATLABCodeTemplate Object from Custom CGT File

This example show how to create a `coder.MATLABCodeTemplate` object from a code generation template (CGT) file that you specify. If the file is not on the MATLAB path, you must specify a full path to the file.

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

newObj =

    MATLABCodeTemplate with properties:

        CGTFile: 'custom_matlabcoder_template.cgt'
```

Version History

Introduced in R2014a

See Also

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

Topics

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

copySetup

Create copy of `cgv.CGV` object

Syntax

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

Description

`cgvObj2 = cgvObj1.copySetup()` creates a copy of a `cgv.CGV` on page 1-43 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 = 'myCGVModel';  
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

`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*

Version History

Introduced in R2007b

See Also

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

createAndAddConceptualArg

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

Syntax

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

Description

`arg = createAndAddConceptualArg(hEntry,argType,Name=Name,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.Tf1COperationEntry;
% .
% .
% .
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'IsSigned',      true, ...
    'WordLength',    32, ...
    'FractionLength', 0);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'IsSigned',      true, ...
    'WordLength',    32, ...
    'FractionLength', 0 );

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    '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.TflC0perationEntry;
% .
% .
% .
% uint8:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'Type', 'uint8');

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

% double:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'Type', 'double' );

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

% complex:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'Type', 'cint16' );

% matrix of complex integers:
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...
    'Name', 'mat_in1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'BaseType', 'cint8', ...
    'DimRange', [2 1; Inf Inf]);

% Fixed-point using binary-point-only scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'Type', 'fixdt(1,32,28)');

% Fixed-point using [slope bias] scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name', 'y1', ...
```

```

    'IOType',          'RTW_IO_OUTPUT', ...
    'Type',            'fixdt(1,16,15,2)');

```

Specify Type Properties for Conceptual Argument

These examples show how to specify types by using several properties of the data type.

```

hEntry = RTW.TflC0perationEntry;
% .
% .
% .
% 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);
```

Specify a Matrix Argument

This example shows how to create an input argument that is a two-dimensional matrix of size 2-by-1 or larger and has base type uint8.

```
hEntry = RTW.TflCOperationEntry;  
% .  
% .  
% .  
createAndAddConceptualArg(hEntry, 'RTW.TflArgMatrix', ...  
    'Name', 'mat_in1', ...  
    'IOType', 'RTW_IO_INPUT', ...  
    'DimRange', [2 1; Inf Inf], ...  
    'BaseType', 'uint8');
```

Specify a Structure Argument

This example shows how to create a conceptual argument that is a structure with elements bus1 and bus2.

```
hEnt = RTW.TflEntry;  
myStruct.Identifier = 'myBus';  
elem1.Identifier = 'bus1';  
elem1.Type = 'int32';  
elem2.Identifier = 'bus2';  
elem2.Type = 'double';  
myStruct.Elements = [elem1, elem2];  
hEnt.createAndAddConceptualArg('RTW.TflArgStruct', 'Name', 'u1', 'StructData', myStruct);
```

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

character vector | string scalar

Argument type to create, specified as a character vector or string scalar. Specify one of these types of arguments.

- 'RTW.TflArgNumeric' — numeric argument
- 'RTW.TflArgMatrix' — matrix argument
- 'RTW.TflArgComplex' — complex argument

- 'RTW.TflArgChar' — character argument
- 'RTW.TflArgVoid' — void argument
- 'RTW.TflArgStruct' — structure argument

Name — Name of the argument to create

character vector | string scalar

Name of the argument to create, specified as a character vector or string scalar. For input arguments, the name must be 'u1', 'u2', or follow the format 'un'. For output arguments, the name must be 'y1', 'y2', or follow the format 'yn'.

Example: 'Name', 'y1'

Example: 'Name', 'u1'

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

name-value pair

Example: 'IOType', 'RTW_IO_INPUT'

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: 'IOType', 'RTW_IO_INPUT'

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'

Type — Data type of the argument

character vector | string scalar

Data type of the argument, specified as a character vector or string scalar. You can specify built-in MATLAB data types such as uint8, boolean, double, and others. You can also specify data types that you create by using the fixdt function, such as fixdt(1,16,2). When you specify the Type, you do not need to specify other properties of the type, such as the signedness or word length.

Example: 'Type', 'uint8'

Example: 'Type', 'fixdt(1,16,2)'

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]`

StructElements — Elements of structure argument

structure

Elements of the structure for a structure argument, specified as a structure in which each field identifies an element of the structure.

Example: 'StructElements',elements

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.

Version History

Introduced in R2007b

R2023b: Specify pre-defined data types by using the Type argument

You can now specify pre-defined data types for the argument by using the `Type` argument. Specify built-in MATLAB data types or fixed-point data types. When you use the `Type` argument, you do not need to specify other properties of the data type such as `IsSigned` or `WordLength`.

See Also

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

createAndAddImplementationArg

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

Syntax

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

Description

`arg = createAndAddImplementationArg(hEntry,argType,Name=Name,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.TfIcOperationEntry;
% .
% .
% .
createAndSetCImplementationReturn(op_entry, 'RTW.TfIcArgNumeric', ...
    'Name',      'y1', ...
    'IOType',    'RTW_IO_OUTPUT', ...
    'IsSigned',  true, ...
    'WordLength', 32, ...
    'FractionLength', 0);

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

createAndAddImplementationArg(op_entry, 'RTW.TfIcArgNumeric',...
    '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.TflC0perationEntry;
% .
% .
% .
% uint8:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'Type', 'uint8');

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

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

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

% complex:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'Type', 'cint16' );

% matrix of complex integers:
createAndAddImplementationArg(hEntry, 'RTW.TflArgMatrix', ...
    'Name', 'mat_in1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'BaseType', 'cint8', ...
    'DimRange', [2 1; Inf Inf]);
```

Specify Type Properties for Implementation Argument

These examples show how to specify types by using several properties of the data type.

```
hEntry = RTW.TflC0perationEntry;
% .
% .
% .
% 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' );

```

Specify a Numeric Constant Argument

This example shows how to add an input argument that is a numeric constant to the implementation function. The actual value of the constant is set during code generation.

```

hEntry = RTW.TflCOperationEntry;
% .
% .
% .
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumericConstant', ...
    'Name',      'fl_in1', ...
    'IOType',    'RTW_IO_INPUT', ...
    'IsSigned',  false, ...
    'WordLength', 32, ...
    'FractionLength', 0, ...
    'Value',     0);

```

Specify a Matrix Argument

This example shows how to create an input argument that is a two-dimensional matrix of size 2-by-1 or larger and has base type uint8.

```

hEntry = RTW.TflCOperationEntry;
% .
% .
% .
createAndAddImplementationArg(hEntry, 'RTW.TflArgMatrix', ...
    'Name',      'mat_in1', ...
    'IOType',    'RTW_IO_INPUT', ...

```

```
'DimRange', [2 1; Inf Inf], ...  
'BaseType', 'uint8');
```

Specify a Structure Argument

This example shows how to create an implementation argument that is a structure with elements `bus1` and `bus2`.

```
hEnt = RTW.TflEntry;  
myStruct.Identifier = 'myBus';  
elem1.Identifier = 'bus1';  
elem1.Type = 'int32';  
elem2.Identifier = 'bus2';  
elem2.Type = 'double';  
myStruct.Elements = [elem1, elem2];  
hEnt.createAndAddImplementationArg('RTW.TflArgStruct', 'Name', 'u1', 'StructData', myStruct);
```

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

character vector | string scalar

Argument type to create, specified as a character vector or string scalar. Specify one of these types of arguments.

- 'RTW.TflArgNumeric' — numeric argument
- 'RTW.TflArgNumericConstant' — numeric constant argument
- 'RTW.TflArgMatrix' — matrix argument
- 'RTW.TflArgComplex' — complex argument
- 'RTW.TflArgChar' — character argument
- 'RTW.TflArgVoid' — void argument
- 'RTW.TflArgStruct' — structure argument

Name — Name of the argument to create

character vector | string scalar

Name of the argument to create, specified as a character vector or string scalar. You may specify:

- 'u1', 'u2', or follow the format 'un' for input arguments
- 'y1', 'y2', or follow the format 'yn' for output arguments
- Other character vector or string scalar names for input or output arguments

Example: 'Name', 'y1'

Example: 'Name', 'u1'

varargin — Name-value pairs that specify the implementation argument

name-value pairs

Example: 'IOType', 'RTW_IO_INPUT'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: 'IOType', 'RTW_IO_INPUT'

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'

Type — Data type of the argument

character vector | string scalar

Data type of the argument, specified as a character vector or string scalar. You can specify built-in MATLAB data types such as `uint8`, `boolean`, `double`, and others. You can also specify data types that you create by using the `fixdt` function, such as `fixdt(1,16,2)`. When you specify the `Type`, you do not need to specify other properties of the type, such as the signedness or word length.

Example: 'Type', 'uint8'

Example: 'Type', 'fixdt(1,16,2)'

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

StructElements — Elements of structure argument

structure

Elements of the structure for a structure argument, specified as a structure in which each field identifies an element of the structure.

Example: 'StructElements',elements

Output Arguments**arg — Handle to the created implementation argument**

handle

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

Version History

Introduced in R2007b

R2023b: Specify pre-defined data types by using the Type argument

You can now specify pre-defined data types for the argument by using the `Type` argument. Specify built-in MATLAB data types or fixed-point data types. When you use the `Type` argument, you do not need to specify other properties of the data type such as `IsSigned` or `WordLength`.

See Also`createAndSetCImplementationReturn`**Topics**

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

createAndSetCImplementationReturn

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

Syntax

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

Description

`arg = createAndSetCImplementationReturn(hEntry,argType,Name=Name,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', ...
    'Type',          'uint8' );

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

% double:
createAndSetCImplementationReturn(hEntry, 'RTW.TfLArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'Type',          'double' );

% boolean:
createAndSetCImplementationReturn(hEntry, 'RTW.TfLArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'Type',          'boolean' );

% complex:
createAndSetCImplementationReturn(hEntry, 'RTW.TfLArgNumeric', ...
    'Name',          'y1', ...
    'IOType',        'RTW_IO_OUTPUT', ...
    'Type',          'cint16' );
```

Specify Type Properties for Operator Implementation

These examples show how to specify types by using several properties of the data type.

```
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' );
```

Specify a Matrix Argument

This example shows how to create a return argument that is a two-dimensional matrix of size 2-by-1 or larger and has base type uint8.

```
hEntry = RTW.TflCOperationEntry;
% .
% .
% .
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgMatrix', ...
    'Name', 'mat_out1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'DimRange', [2 1; Inf Inf], ...
    'BaseType', 'uint8');
```

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'

Name — Name of the argument to create

character vector | string scalar

Name of the argument to create, specified as a character vector or string scalar.

Example: 'Name', 'y1'

Example: 'Name', 'argOut'

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

name-value pairs

Example: 'IOType', 'RTW_IO_OUTPUT'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: 'IOType', 'RTW_IO_OUTPUT'

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'

Type — Data type of the argument

character vector | string scalar

Data type of the argument, specified as a character vector or string scalar. You can specify built-in MATLAB data types such as `uint8`, `boolean`, `double`, and others. You can also specify data types that you create by using the `fixdt` function, such as `fixdt(1,16,2)`. When you specify the `Type`, you do not need to specify other properties of the type, such as the signedness or word length.

Example: 'Type', 'uint8'

Example: 'Type', 'fixdt(1,16,2)'

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.

Version History

Introduced in R2007b

R2023b: Specify pre-defined data types by using the Type argument

You can now specify pre-defined data types for the argument by using the `Type` argument. Specify built-in MATLAB data types or fixed-point data types. When you use the `Type` argument, you do not need to specify other properties of the data type such as `IsSigned` or `WordLength`.

See Also

`createAndAddImplementationArg`

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

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 <code>int32*</code>, 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 <code>fixdt(1,32,*)</code>, wildcard.
~	Based on the position of the symbol, slopes or bias must be the same across data types.
;	Separates dimension ranges. For example, <code>[1 10; 1 100]</code> 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	<code>double y1 = sin(double u1)</code>	<code>double y1 = mySin(double u1)</code>
Derive implementation argument data types from conceptual specification	<code>double y1 = sin(double u1)</code>	<code>y1 = mySin(u1)</code>
Derive implementation arguments and data types from conceptual specification	<code>double y1 = sin(double u1)</code>	<code>mySin</code>
Change data type	<code>single y1 = sin(single u1)</code>	<code>double y1 = mySin(double u1)</code>
Reorder arguments	<code>double y1 = atan2(double u1, double u2)</code>	<code>y1 = myAtan(u2, u1)</code>
Specify column vector arguments	<code>double y1 = sin(double u1[10])</code>	<code>double y1 = mySin(double* u1)</code>
Specify column vector arguments and dimension range	<code>double y1[1 100; 1 100] = sin(double u1[1 100; 1 100])</code>	<code>mySin(double* u1, double* y1)</code>
Remap return value as output argument	<code>double y1 = sin(double u1)</code>	<code>mySin(double u1, double* y1)</code>
Specify fixed-point data types	<code>fixdt(1,16,3) y1 = sin(fixdt(1,16,3) u1)</code>	<code>int16 y1 = mySin(int16 u1)</code>
Specify fixed-point data types and set CheckSlope to false, CheckBias to true, and Bias to 0	<code>fixdt(1,16,*) y1 = sin(fixdt(1,16,*) u1)</code>	<code>int16 y1 = mySin(int16 u1)</code>
Specify fixed-point data types and set SlopesMustBeTheSame to true, CheckSlope to false, CheckBias to true, and Bias to 0	<code>fixdt(1,16,~) y1 = sin(fixdt(1,16,~) u1)</code>	<code>int16 y1 = mySin(int16 u1)</code>

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	<code>fixdt(1,16,~,~) y1 = sin(fixdt(1,16,~,~) u1)</code>	<code>int16 y1 = mySin(int16 u1)</code>
Specify multiple output arguments	<code>[double y1 double y2] = foo(double u1, double u2)</code>	<code>double y1 = myFoo(double u1, double u2, double* y2)</code>
Operator Code Replacement Syntax		
Typical	<code>int16 y1 = int16 u1 + int16 u2</code>	<code>int16 y1 = myAdd(int16 u1, int16 u2)</code>
Specify fixed-point data types	<code>fixdt(1,16,3) y1 = fixdt(1,16,3) u1 + fixdt(1,16,3) u2</code>	<code>int16 y1 = myAdd(int16 u1, int16 u2)</code>
Specify fixed-point data types and set CheckSlope to false, CheckBias to true, and Bias to 0	<code>fixdt(1,16,*) y1 = fixdt(1,16,*) u1 + fixdt(1,16,*) u2</code>	<code>int16 y1 = myAdd(int16 u1, int16 u2)</code>
Specify fixed-point data types, wildcard, slopes must be the same, and zero bias	<code>fixdt(1,16,~,0) y1 = fixdt(1,16,~,0) u1 + fixdt(1,16,~,0) u2</code>	<code>int16 y1 = myAdd(int16 u1, int16 u2)</code>
Typecast	<code>int16 y1 = int8 u1</code>	<code>int16 y1 = myCast(int8 u1)</code>
Shift	<code>int16 y1 = int16 u1 << int16 u2 int16 y1 = int16 u1 >> int16 u2 int16 y1 = int16 u1 .>> int16 u2</code>	<code>int16 y1 = myShiftLeft(int16 u1, int16 u2) int16 y1 = myShiftRightArithmetic(int16 u1, int16 u2) int16 y1 = myShiftRightLogical(int16 u1, int16 u2)</code>
Specify relational operator	<code>bool y1 = int16 u1 < int16 u2</code>	<code>bool y1 = myLessThan(int6 u1, int16 u2)</code>
Specify multiplication and division	<code>int32 y1 = int32 u1 * in32 u2 / in32 u3</code>	<code>int32 y1 = myMultDiv(int32 u1, int32 u2, int32 u3)</code>
Specify matrix multiplication	<code>double y1[10][10] = double u1[10] [10] * double u2[10][10]</code>	<code>myMult(double* u1, double* u2, double* y1)</code>

Type of Replacement	Conceptual Syntax	Implementation Syntax
Specify element-wise matrix multiplication	<code>double y1[10][10] = double u1[10][10] .* double u2[10][10]</code>	<code>myMult(double* u1, double* u2, double* y1)</code>
Specify matrix multiplication with transpose of an input argument	<code>double y1[10][10] = double u1[10][10]' * double u2[10][10]</code>	<code>myMult(double* u1, double* u2, double* y1)</code>
Specify matrix multiplication with Hermitian of an input argument	<code>cdouble y1[10][10] = cdouble u1[10][10]' * cdouble u2[10][10]</code> <code>cdouble y1[10][10] = cdouble u1[10][10] * cdouble u2[10][10]'</code>	<code>myMult(cdouble* u1, cdouble* u2, cdouble* y1)</code>
Specify left matrix division	<code>double y1[10][10] = double u1[10][10] / double u2[10][10]</code>	<code>myLeftDiv(double* u1, double* u2, double* y1)</code>
Specify right matrix division	<code>double y1[10][10] = double u1[10][10] \ double u2[10][10]</code>	<code>myRightDiv(double* u1, double* u2, double* y1)</code>

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`.

```
setTflCOperationEntryParameters(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 1-244 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 1-244 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`.

Version History

Introduced in R2015a

See Also

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

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

createToleranceFile

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', 'folderPathForPl_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

“Code generation folder” 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'}.

Version History

Introduced in R2016b

R2023b: Changed

The `crossReleaseImport` function supports the import of generated code from only the previous eight releases. If you use the function to import generated code from earlier releases, the function produces an error.

R2023a: Warns

Warns starting in R2023a

The function issues a warning that it will, in future releases, support the import of generated code from only the previous eight releases. You can use the `crossReleaseImport` function to import generated code from R2010a and later releases.

See Also

`sharedCodeUpdate`

Topics

“Cross-Release Shared Utility Code Reuse”

“Cross-Release Code Integration”

display

Namespace: coder.profile

Generate message that describes how to open code execution profiling report

Syntax

```
myExecutionProfile.display
```

Description

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

Examples

Display Code Execution Profiling Report

To generate a message that describes how to open the code execution profiling report, use the `display` function and the `myExecutionProfile` workspace variable.

```
myExecutionProfile.display
```

Input Arguments

myExecutionProfile — Variable specifies annotation

workspace variable

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

Example: `myExecutionProfile`

Version History

Introduced in R2011a

See Also

`report`

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

displayReport

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 with CGV”

coder.MATLABCodeTemplate.emitSection

Class: coder.MATLABCodeTemplate

Namespace: 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.setTokenValue](#) |
 [coder.MATLABCodeTemplate.getTokenValue](#) |
 [coder.MATLABCodeTemplate.getCurrentTokens](#)

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 Library”.

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`

Version History

Introduced in R2010a

See Also

`registerCPPFunctionEntry` | `setNameSpace`

Topics

["Math Function Code Replacement"](#)

["Define Code Replacement Library Optimizations"](#)

["Code You Can Replace from MATLAB Code"](#)

["Code You Can Replace From Simulink Models"](#)

excludeCheck

Namespace: `rtw.codegenObjectives`

Exclude check from code generation objective

Syntax

`excludeCheck(objective, checkID)`

Description

`excludeCheck(objective, checkID)` excludes the specified check from the Code Generation Advisor when you specify the objective. When you select multiple objectives, if you specify an additional objective that includes this check as a higher priority objective, the Code Generation Advisor includes this check.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the Traceability objective.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace', 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConvertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');

% Exclude the inherited check for the software environment
```

```
excludeCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');  
  
%Register the objective  
register(obj);  
  
end
```

Input Arguments

objective — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

checkID — Identifier of check

character vector | string scalar

Identifier of check that you want to exclude, specified as a character vector or string scalar.

Example: `'mathworks.codegen.CodeInstrumentation'`

Version History

Introduced in R2009a

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.TfTable;
```

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]  
        ApplyFullPrecisionForLinearInterpolation: [1x1 coder.algorithm.parameter.ApplyFullPrecisionForLinearInterpolation]  
        RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]  
        SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]  
        SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]  
        BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]  
        BeginIndexSearchUsingPreviousIndexResult: [1x1 coder.algorithm.parameter.BeginIndexSearchUsingPreviousIndexResult]
```

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`.
- 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`.

```
tableEntry = RTW.TflCFunctionEntry;

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`.

Version History

Introduced in R2015a

See Also

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

Topics

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

`coder.MATLABCodeTemplate.getCurrentTokens`

Class: `coder.MATLABCodeTemplate`

Namespace: `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.setTokenValue` | `coder.MATLABCodeTemplate.getTokenValue`
| `coder.MATLABCodeTemplate.emitSection`

Topics

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

“Code Generation Template Files for MATLAB Code”

getCoderExecutionProfile

Namespace: `coder.profile.mc`

Extract execution-time profile for code generated from MATLAB function (MATLAB code generation)

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.

Examples

Get Coder Execution Profile

To get the code generator execution profile for a MATLAB function, use the `getCoderExecutionProfile` function.

```
myExecutionProfile = getCoderExecutionProfile('myMATLABFunction');
```

Input Arguments

myMATLABFunction — Function whose generate code is profiled
function

The `myMATLABFunction` is the MATLAB function whose generate code is profiled for execution-time.

Example: `myMATLABFunction`

Output Arguments

myExecutionProfile — Workspace variable that contains the execution-time profile
workspace variable

The `myExecutionProfile` is a workspace variable that contains the execution-time profile of the code generated from your MATLAB function.

Version History

Introduced in R2014b

See Also

TimerTicksPerSecond | Sections | report

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

Name

Namespace: `coder.profile`

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.

Examples

Get Name of Profiled Code Section

To get the name that identifies the profiled code section, use the `Name` property of the `NthSectionProfile` object.

```
SectionName = NthSectionProfile.Name;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

SectionName — Profiled section name

section name

Name that identifies profiled code section.

Version History

Introduced in R2012b

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` |

MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

Name

Namespace: `coder.profile.mc`

Get name of profiled code section (MATLAB code generation)

Syntax

`SectionName = NthSectionProfile.Name`

Description

`SectionName = NthSectionProfile.Name` returns the name that identifies the profiled code section.

Examples

Get Name of Profiled Code Section

To get the name that identifies the profiled code section, use the `Name` property of the `NthSectionProfile` object.

```
SectionName = NthSectionProfile.Name;
```

Input Arguments

NthSectionProfile — `ExecutionTimeSection` object

`ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile.MaximumExecutionTimeInTicks`

Output Arguments

SectionName — Profiled section name

section name

The `SectionName` is the name that identifies profiled code section.

Version History

Introduced in R2012b

See Also

`getCoderExecutionProfile` | `Sections` | `TimerTicksPerSecond` | `report` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` |

ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |
SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum |
TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

NumCalls

Namespace: `coder.profile`

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.

Examples

Get Number of Calls to Profiled Section

To get the total number of calls to the profiled code section over the entire simulation, use the `NumCalls` property of the `NthSectionProfile` object.

```
TotalNumCalls = NthSectionProfile.NumCalls;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalNumCalls — Total number of calls

integer

The total number of calls to the profiled section of code.

Version History

Introduced in R2012b

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `Number` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` |

MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

NumCalls

Namespace: `coder.profile.mc`

Total number of calls to profiled code section (MATLAB code generation)

Syntax

`TotalNumCalls = NthSectionProfile.NumCalls`

Description

`TotalNumCalls = NthSectionProfile.NumCalls` returns the total number of calls to the profiled code section over the entire execution.

Examples

Get Number of Calls to Profiled Section

To get the total number of calls to the profiled code section over the entire simulation, use the `NumCalls` property of the `NthSectionProfile` object.

```
TotalNumCalls = NthSectionProfile.NumCalls;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalNumCalls — Total number of calls

integer

The total number of calls to the profiled section of code.

Version History

Introduced in R2012b

See Also

`getCoderExecutionProfile` | `Sections` | `TimerTicksPerSecond` | `report` | `Name` | `Number` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` |

MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

getOutputData

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”

getReportData

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

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

Namespace: `coder.profile`

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.

Examples

Get Number of Profiled Code Section

To get the number that identifies the profiled code section, use the `Number` property of the `NthSectionProfile` object.

```
SectionNumber = NthSectionProfile.Number;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

SectionNumber — Number of profiled section

integer

The `SectionNumber` is the number of the profiled code section.

Version History

Introduced in R2012b

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` |

MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

Number

Namespace: `coder.profile.mc`

Get number that uniquely identifies profiled code section (MATLAB code generation)

Syntax

`SectionNumber = NthSectionProfile.Number`

Description

`SectionNumber = NthSectionProfile.Number` returns a number that uniquely identifies the profiled code section.

Examples

Get Number of Profiled Code Section

To get the number that identifies the profiled code section, use the `Number` property of the `NthSectionProfile` object.

```
SectionNumber = NthSectionProfile.Number;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

SectionNumber — Number of profiled section

integer

The `SectionNumber` is the number of the profiled code section.

Version History

Introduced in R2012b

See Also

`getCoderExecutionProfile` | `Sections` | `TimerTicksPerSecond` | `report` | `Name` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` |

MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

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'

Version History

Introduced in R2006b

See Also

Topics

“Cross-Release Code Integration”

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;
```

Open the model and specify the PIL simulation mode.

```
openExample('ecoder/SILPILVerificationExample', ...
    supportingFile='SILTopModel.slx')
set_param('SILTopModel','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('SILTopModel','TargetHWDeviceType',...
    'Intel->x86-64 (Windows64)');
set_param('SILTopModel','TargetLongLongMode',true);
```

Run `piltest`.

```
piltest('SILTopModel', 'ConfigParam', {'ProdLongLongMode'} )
```

Input Arguments

config — Configuration set, configuration reference, or model

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.

Version History

Introduced in R2016b

See Also

`ConfigSet` | `Simulink.ConfigSetRef`

Topics

“Create PIL Target Connectivity Configuration for Simulink”

“SIL and PIL Simulations”

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.

Version History

Introduced in R2016b

See Also

Topics

“Create PIL Target Connectivity Configuration for MATLAB”

“PIL Execution of Code Generated for a Kalman Estimator”

Sections

Namespace: `coder.profile`

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 `Nth` profiled code section.

`numberOfSections=length(myExecutionProfile.Sections)` returns the number of code sections for which profile data is available.

Examples

Get Nth Section Profile

Get the `coder.profile.ExecutionTimeSection` object for the `Nth` profiled code section.

```
NthSectionProfile=myExecutionProfile.Sections(N);
```

Get Number of Profiled Code Sections

Get the number of code sections for which profile data is available.

```
numberOfSections=length(myExecutionProfile.Sections);
```

Input Arguments

myExecutionProfile — Workspace variable

workspace variable

`myExecutionProfile` is a workspace variable generated by a simulation.

Example: `myExecutionProfile`

N — Index of code section

index of code section

Index of code section for which profile data is required

Example: `N`

Output Arguments

NthSectionProfile — profile information object

profile information object

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

number of code sections

The `numberOfSections` is the number of code sections with profile data.

Version History

Introduced in R2012b

See Also

`TimerTicksPerSecond` | `display` | `report`

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

Sections

Namespace: `coder.profile.mc`

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections (MATLAB code generation)

Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)  
numberOfSections = length(myExecutionProfile.Sections)
```

Description

`NthSectionProfile = myExecutionProfile.Sections(N)` returns an `coder.profile.ExecutionTimeSection` object for the `Nth` profiled code section.

`numberOfSections = length(myExecutionProfile.Sections)` returns the number of code sections for which profile data is available.

Examples

Get Nth Section Profile

Get the `coder.profile.ExecutionTimeSection` object for the `Nth` profiled code section.

```
NthSectionProfile = myExecutionProfile.Sections(N);
```

Get Number of Profiled Code Sections

Get the number of code sections for which profile data is available.

```
numberOfSections = length(myExecutionProfile.Sections)
```

Input Arguments

myExecutionProfile — Workspace variable created by `getCoderExecutionProfile`
workspace variable

The `myExecutionProfile` is a workspace variable that you create by using the `getCoderExecutionProfile` function.

Example: `myExecutionProfile`

N — Index of code section

index of code section

Index of code section for which profile data is required.

Example: N

Output Arguments

NthSectionProfile — profile information object

profile information object

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

number of code sections

Number of code sections with profile data

Version History

Introduced in R2012b

See Also

`getCoderExecutionProfile` | `report` | `TimerTicksPerSecond`

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

coder.profile.ExecutionStackSet

Aggregate stack usage profiling results from multiple SIL, PIL, or external mode simulations

Description

Use a `coder.profile.ExecutionStackSet` object for the aggregation of stack usage profiling results from multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of a model. Then, you can use the Code Profile Analyzer to identify the most demanding execution for each task in the generated code. You can also identify the code or model path that produces the most demanding execution.

Creation

Syntax

```
myResults = coder.profile.ExecutionStackSet(modelName);
```

Description

`myResults = coder.profile.ExecutionStackSet(modelName);` creates an object for storing stack usage profiling results produced by multiple SIL or PIL simulations of `modelName`.

Input Arguments

modelName — Model name

character vector | string

Name of simulation model.

Properties

ResultsPerTest — Executions considered

5 (default) | integer scalar

Specifies the number of most demanding task executions to consider for each model simulation.

Example: `resultsObject.ResultsPerTest=10;`

Object Functions

<code>add</code>	Aggregate stack usage profiles
<code>addDetails</code>	Add text items to Code Profile Analyzer display or code profiling report
<code>get</code>	Extract specific execution-time profile from aggregate of profiles
<code>remove</code>	Remove execution-time profile from aggregate of profiles

Examples

Aggregate Stack Usage Profiles and Identify Most Demanding Task Execution

For an example workflow that uses the object, see “Aggregate Stack Usage Profiles to Identify Most Demanding Execution”.

Version History

Introduced in R2023b

See Also

Topics

“View and Compare Stack Usage Metrics”

“Automate Execution-Time and Stack Usage Profiling of MATLAB Function” on page 1-321

add

Namespace: `coder.profile`

Aggregate stack usage profiles

Syntax

```
myResultsObject.add(simulationName, anExecutionProfile);
```

Description

`myResultsObject.add(simulationName, anExecutionProfile)`; adds the stack usage metrics in `anExecutionProfile` to `myResultsObject` and associates the added data with `simulationName`. Use this function to create an aggregate of profiles from multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of a model.

Examples

Aggregate Stack Usage Profiles

For an example that uses the function, see [Aggregate Stack Usage Profiles and Identify Most Demanding Task Execution](#) on page 1-302.

Input Arguments

simulationName — Simulation name

character vector | string

Name for simulation that produced stack usage metrics.

Example: `resultsObject.add('Test 1', simOut.stackProfile);`

anExecutionProfile — Workspace variable

`coder.profile.ExecutionStack` object

Workspace variable that contains stack usage metrics from the model simulation.

Example: `resultsObject.add('Test 1', simOut.stackProfile);`

myResultsObject — Aggregate of stack usage profiles

`coder.profile.ExecutionStackSet` object

Object that contains aggregate of stack usage profiles produced by multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of a model.

Example: `resultsObject.add('Run 1', simOut.stackProfile);`

Version History

Introduced in R2023b

See Also

`coder.profile.ExecutionStackSet` | `get` | `remove`

Topics

Aggregate Stack Usage Profiles and Identify Most Demanding Task Execution on page 1-302

addDetails

Namespace: `coder.profile`

Add text items to Code Profile Analyzer display or code profiling report

Syntax

```
myResultsObject.addDetails(itemName,itemText);
```

Description

`myResultsObject.addDetails(itemName,itemText)`; adds a text item in the **Cumulative Results > Test Execution Information** section of the Code Profile Analyzer app. If you create a PDF report, the first section of the report also displays the text item. Use this function to display additional information about your stack usage analysis.

Examples

Add Text Execution Information

For an example that uses the function, see “Customize Automated Code Profiling Using Hooks” on page 1-324.

Input Arguments

itemName — Item name

character vector | string scalar

Text to display for item.

Example: `resultsObj.addDetails("Item1","Description for Item1");`

itemText — Item text

character vector | string scalar

Text to display for item.

Example: `resultsObj.addDetails("Item1","Description for Item1");`

myResultsObject — Aggregate of execution-time profiles

`coder.profile.ExecutionStackSet` object

Object that contains aggregate of stack usage profiles produced by multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.

Version History

Introduced in R2025a

See Also

`coder.profile.test.runTests` | `coder.profile.ExecutionStackSet`

Topics

“Aggregate Stack Usage Profiles to Identify Most Demanding Execution”

“Customize Automated Code Profiling Using Hooks” on page 1-324

get

Namespace: `coder.profile`

Extract specific stack usage profile from aggregate of profiles

Syntax

```
anExecutionProfile = myResultsObject.get(simulationName)
```

Description

`anExecutionProfile = myResultsObject.get(simulationName)` returns a workspace variable that contains stack usage metrics for `simulationName`.

Examples

Extract Stack Usage Profile From Aggregate

For an example that uses the function, see [Aggregate Stack Usage Profiles and Identify Most Demanding Task Execution](#) on page 1-302.

Input Arguments

simulationName — Simulation name

character vector | string

Name of simulation that produced stack usage profile.

Example: `executionProfileForRun2 = resultsObject.get('Run 2')`

myResultsObject — Aggregate of stack usage profiles

`coder.profile.ExecutionStackSet` object

Object that contains aggregate of stack usage profiles produced by multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of a model.

Example: `executionProfileForRun2 = resultsObject.get('Run 2')`

Output Arguments

anExecutionProfile — Workspace variable

`coder.profile.ExecutionStack` object

Workspace variable that contains stack usage profiling data from named simulation.

Version History

Introduced in R2023b

See Also

`coder.profile.ExecutionStackSet` | [add](#) | [remove](#)

Topics

[Aggregate Stack Usage Profiles and Identify Most Demanding Task Execution on page 1-302](#)

remove

Namespace: `coder.profile`

Remove stack usage profile from aggregate of profiles

Syntax

```
myResultsObject.remove(simulationName);
```

Description

`myResultsObject.remove(simulationName)`; removes stack usage metrics associated with `simulationName` from the aggregate of profiles in `myResultsObject`.

Examples

Remove Stack Usage Profile From Aggregate

If you want to remove specific simulation results from a `coder.profile.ExecutionStackSet` object, use the `remove` function.

For example:

```
resultsObject.remove('Run 3')
```

Input Arguments

simulationName — Simulation name

character vector | string

Name of simulation that produced stack usage profile.

Example: `resultsObject.remove('Run 3');`

myResultsObject — Aggregate of stack usage profiles

`coder.profile.ExecutionStackSet` object

Object that contains aggregate of stack usage profiles produced by multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of a model.

Example: `resultsObject.remove('Run 3');`

Version History

Introduced in R2023b

See Also

`coder.profile.ExecutionStackSet` | `add` | `get`

Topics

Aggregate Stack Usage Profiles and Identify Most Demanding Task Execution on page 1-302

coder.profile.ExecutionTimeSet

Aggregate code profiling results from multiple SIL, PIL, or external mode simulations

Description

Use a `coder.profile.ExecutionTimeSet` object for the aggregation of code profiling results from multiple software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulations of a model. Then, you can use the Code Profile Analyzer to identify the most demanding execution for each task in generated code. You can also identify the code or model path that produces the most demanding execution.

Creation

Syntax

```
myResults = coder.profile.ExecutionTimeSet(modelName);
```

Description

`myResults = coder.profile.ExecutionTimeSet(modelName);` creates an object for storing execution-time profiling results produced by multiple software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulations of `modelName`.

Input Arguments

modelName — Model name

character vector | string

Name of simulation model.

Properties

ResultsPerTest — Executions considered

5 (default) | integer scalar

Specifies the number of longest task executions to consider for each model simulation.

Example: `resultsObject.ResultsPerTest=10;`

Object Functions

<code>add</code>	Aggregate execution-time profiles
<code>addDetails</code>	Add text items to Code Profile Analyzer display or code profiling report
<code>get</code>	Extract specific execution-time profile from aggregate of profiles
<code>remove</code>	Remove execution-time profile from aggregate of profiles

Examples

Aggregate Execution-Time Profiles and Identify Worst-Case Task Execution

For an example workflow that uses the object, see [Aggregate Execution-Time Profiles to Identify Worst Execution](#).

Version History

Introduced in R2023b

See Also

Topics

[“View and Compare Code Execution Times”](#)

add

Namespace: `coder.profile`

Aggregate execution-time profiles

Syntax

```
myResultsObject.add(simulationName, anExecutionProfile);
```

Description

`myResultsObject.add(simulationName, anExecutionProfile)`; adds the execution-time metrics in `anExecutionProfile` to `myResultsObject` and associates the added data with `simulationName`. Use this function to create an aggregate of profiles from multiple software-in-the-loop (SIL), processor-in-the-loop (PIL), or external mode simulations of a model.

Examples

Aggregate Execution-Time Profiles

For an example that uses the function, see [Aggregate Execution-Time Profiles and Identify Worst-Case Task Execution](#).

Input Arguments

simulationName — Simulation name

character vector | string

Name for simulation that produced execution-time metrics.

Example: `resultsObject.add('Run 1', simOut.executionProfile);`

anExecutionProfile — Workspace variable

`coder.profile.ExecutionTime` object

Workspace variable that contains execution-time metrics from the model simulation.

Example: `resultsObject.add('Run 1', simOut.executionProfile);`

myResultsObject — Aggregate of execution-time profiles

`coder.profile.ExecutionTimeSet` object

Object that contains aggregate of execution-time profiles produced by multiple software-in-the-loop (SIL), processor-in-the-loop (PIL), or external mode simulations of a model.

Example: `resultsObject.add('Run 1', simOut.executionProfile);`

Version History

Introduced in R2023b

See Also

`coder.profile.ExecutionTimeSet` | `get` | `remove`

Topics

Aggregate Execution-Time Profiles to Identify Worst Execution

addDetails

Namespace: `coder.profile`

Add text items to Code Profile Analyzer display or code profiling report

Syntax

```
myResultsObject.addDetails(itemName,itemText);
```

Description

`myResultsObject.addDetails(itemName,itemText)`; adds a text item in the **Cumulative Results > Test Execution Information** section of the Code Profile Analyzer app. If you create a PDF report, the first section of the report also displays the text item. Use this function to display additional information about your execution-time analysis.

Examples

Add Text Execution Information

For an example that uses the function, see “Customize Automated Code Profiling Using Hooks” on page 1-324.

Input Arguments

itemName — Item name

character vector | string scalar

Text to display for item.

Example: `resultsObj.addDetails("Item1","Description for Item1");`

itemText — Item text

character vector | string scalar

Text to display for item.

Example: `resultsObj.addDetails("Item1","Description for Item1");`

myResultsObject — Aggregate of execution-time profiles

`coder.profile.ExecutionTimeSet` object

Object that contains aggregate of execution-time profiles produced by multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.

Version History

Introduced in R2025a

See Also

`coder.profile.test.runTests` | `coder.profile.ExecutionTimeSet`

Topics

“Aggregate Execution-Time Profiles to Identify Worst Execution”

“Customize Automated Code Profiling Using Hooks” on page 1-324

get

Namespace: `coder.profile`

Extract specific execution-time profile from aggregate of profiles

Syntax

```
anExecutionProfile = myResultsObject.get(simulationName)
```

Description

`anExecutionProfile = myResultsObject.get(simulationName)` returns a workspace variable that contains execution-time metrics for `simulationName`.

Examples

Extract Profile From Aggregate

For an example that uses the function, see [Aggregate Execution-Time Profiles and Identify Worst-Case Task Execution](#).

Input Arguments

simulationName — Simulation name

character vector | string

Name of simulation that produced execution-time profile.

Example: `executionProfileForRun2 = resultsObject.get('Run 2')`

myResultsObject — Aggregate of execution-time profiles

`coder.profile.ExecutionTimeSet` object

Object that contains aggregate of execution-time profiles produced by multiple software-in-the-loop (SIL), processor-in-the-loop (PIL), or external mode simulations of a model.

Example: `executionProfileForRun2 = resultsObject.get('Run 2')`

Output Arguments

anExecutionProfile — Workspace variable

`coder.profile.ExecutionTime` object

Workspace variable that contains execution profiling data from named simulation.

Version History

Introduced in R2023b

See Also

`coder.profile.ExecutionTimeSet` | `add` | `remove`

Topics

Aggregate Execution-Time Profiles to Identify Worst Execution

remove

Namespace: `coder.profile`

Remove execution-time profile from aggregate of profiles

Syntax

```
myResultsObject.remove(simulationName);
```

Description

`myResultsObject.remove(simulationName)`; removes execution-time metrics associated with `simulationName` from the aggregate of profiles in `myResultsObject`.

Examples

Remove Profile From Aggregate

For an example that uses the function, see [Aggregate Execution-Time Profiles and Identify Worst-Case Task Execution](#).

Input Arguments

simulationName — Simulation name

character vector | string

Name of simulation that produced execution-time profile.

Example: `resultsObject.remove('Run 3');`

myResultsObject — Aggregate of execution-time profiles

`coder.profile.ExecutionTimeSet` object

Object that contains aggregate of execution-time profiles produced by multiple software-in-the-loop (SIL), processor-in-the-loop (PIL), or external mode simulations of a model.

Example: `resultsObject.remove('Run 3');`

Version History

Introduced in R2023b

See Also

`coder.profile.ExecutionTimeSet` | `add` | `get`

Topics

[Aggregate Execution-Time Profiles to Identify Worst Execution](#)

coder.profile.test.runTests

Namespace: `coder.profile.test`

Automate execution-time or stack usage analysis of code generated from Simulink models or MATLAB functions

Syntax

```
resultsSet = coder.profile.test.runTests(testComponent)
resultsSet = coder.profile.test.runTests(testComponent,CodeMetric="stack")
resultsSet = coder.profile.test.runTests(testComponent,TestFile="myTestFile")
resultsSet = coder.profile.test.runTests(
testComponent,MATLABConfig=objConfig,MATLABInputs=inputSource)
[resultsSet secondResultsSet] = coder.profile.test.runTests(____
,OverrideSettings=keyValueMapping)
resultsSet = coder.profile.test.runTests(____,Name=Value)
```

Description

`resultsSet = coder.profile.test.runTests(testComponent)` runs software-in-the-loop (SIL) or processor-in-the-loop (PIL) tests of the Simulink model, `testComponent`, by using inputs provided by the model. The function runs in two stages. First, the function runs the tests with only task profiling enabled. Then, the function runs the most demanding tests again with detailed function profiling enabled.

In the first stage, `runTests` captures the execution times of tasks for the given inputs. In the second stage, `runTests` identifies how code is executed. For example, it generates a visualization of the function-call stack. `runTests` does not run detailed profiling in the first stage because instrumentation overhead can be significant and invalidate the acquired time measurements.

The function stores execution-time results in the `resultsSet` object. Use the Code Profile Analyzer app to analyze the results.

For information about importing test cases into a model, see “Overview of Signal Loading Techniques” and “Load Input Data for Basic Test Cases”.

`resultsSet = coder.profile.test.runTests(testComponent,CodeMetric="stack")` runs software-in-the-loop (SIL) or processor-in-the-loop (PIL) tests of the Simulink model and stores stack usage results in the `resultsSet` object. For stack usage analysis, there is only one stage because instrumentation does not affect the results. Use the Code Profile Analyzer app to analyze the results.

`resultsSet = coder.profile.test.runTests(testComponent,TestFile="myTestFile")` runs software-in-the-loop (SIL) or processor-in-the-loop (PIL) tests of the model by using test cases in the Simulink Test file *myTestFile*. As explained previously, the function runs in two stages to reduce the impact of instrumentation overhead.

If the test cases cover system requirements and exercise as many as possible code paths, you can use the function and the Code Profile Analyzer app to identify and analyze performance hotspots in the generated code.

`resultsSet = coder.profile.test.runTests(testComponent, MATLABConfig=objConfig, MATLABInputs=inputSource)` runs software-in-the-loop (SIL) or processor-in-the-loop (PIL) tests of the MATLAB function, *testComponent*, by using inputs provided by *inputSource*.

objConfig is a MATLAB configuration object for code generation.

`[resultsSet secondResultsSet] = coder.profile.test.runTests(____, OverrideSettings=keyValueMapping)` runs two SIL or PIL tests using *keyValueMapping*, a Map object that maps model configuration parameters to pairs of values. For the first test, the function uses the first value of each pair. For the second test, the function uses the second value of each pair. The function returns two sets of results, *resultsSet* and *secondResultsSet*.

`resultsSet = coder.profile.test.runTests(____, Name=Value)` runs software-in-the-loop (SIL) or processor-in-the-loop (PIL) tests of the Simulink model or MATLAB function using one or more name-value arguments.

Examples

Use `coder.profile.test.runTests` to Identify Hotspots

You can use `coder.profile.test.runTests` and the Code Profile Analyzer app to identify and analyze performance hotspots in generated code.

For an example, see [Identify Hotspots in Generated Code](#).

Automate Execution-Time and Stack Usage Profiling of MATLAB Function

This example shows how you can perform execution-time and stack usage profiling of code generated from a MATLAB function.

Open the example that contains the required files.

```
openExample('ecoder/KalmanFilterExample')
```

Load the inputs.

```
load position.mat
numPts = 300;
inputs = cell(1, numPts);
for idx = 1: numPts
    inputs{idx} = {position(:,idx)};
end
```

Create a MATLAB configuration object.

```
config = coder.config('lib');
config.VerificationMode = 'SIL';
config.EnableOpenMP = false;
```

Run tests with execution-time and stack usage profiling enabled.

```
myFcn = 'kalman01';
```

```
resTime = coder.profile.test.runTests(myFcn, ...  
                                     MATLABConfig=config, ...  
                                     MATLABInputs=inputs, ...  
                                     NumIterations=5);  
  
resStack = coder.profile.test.runTests(myFcn, ...  
                                       CodeMetric="stack", ...  
                                       MATLABConfig=config, ...  
                                       MATLABInputs=inputs);
```

To view execution-time results in the Code Profile Analyzer app, run:

```
coder.profile.show(resTime);
```

The **Cumulative Results** tab displays the **Task Summary** and **Test Details** views. When you select a function in the **Task Summary** view, the **Test Details** view lists the SIL tests performed in order of execution time.

Cumulative Results				Summary	
Task Summary				Summary	
Task	Period (s)	Offset (s)	Longest Execution Time	Profiled component:	kalman01
kalman01	10	0	8.2015e+03	Analysis type:	Fully Automated
				Environment:	MATLAB Coder
				Number of tests:	20
				Number of tasks:	1
				Time created:	19-Dec-2024 14:10:10
Test Details					
Test Name	Longest Execution Time	Simulation Time	Average Execution Time	Calls	
Inputs 4 (run 2)	8.2015e+03	1	8.2015e+03	1	
Inputs 9 (run 1)	1.4720e+03	1	1.4720e+03	1	
Inputs 10 (run 1)	1.3218e+03	1	1.3218e+03	1	
Inputs 1 (run 1)	1.2018e+03	1	1.2018e+03	1	
Inputs 1 (run 2)	1.2017e+03	1	1.2017e+03	1	
Inputs 3 (run 1)	1.2417e+03	1	1.2417e+03	1	
Inputs 3 (run 2)	1.2217e+03	1	1.2217e+03	1	
Inputs 5 (run 1)	1.2217e+03	1	1.2217e+03	1	
Inputs 2 (run 1)	1.1816e+03	1	1.1816e+03	1	
Inputs 8 (run 1)	1.1316e+03	1	1.1316e+03	1	

For more information about analyzing execution-time results, see “Automate Identification of Hotspots in Generated Code”.

To view stack usage results in the Code Profile Analyzer app, run:

```
coder.profile.show(resStack);
```

The **Cumulative Results** tab displays the **Task Summary** and **Test Details** views. If you select kalman01 in the **Task Summary** view, the **Test Details** view shows constant stack usage for the SIL tests.

Cumulative Results				Summary	
Task Summary				Summary	
Task	Period (s)	Offset (s)	Highest Stack Usage (bytes)	Profiled component:	kalman01
kalman01	10	0	1520	Environment:	MATLAB Coder
				Number of tests:	10
				Number of tasks:	1
				Time created:	19-Dec-2024 14:11:41
Test Details					
Test Name	Highest Stack Usage (bytes)	Highest Stack Usage At	Average Stack Usage (bytes)	Calls	
Inputs 1	1520	1	1520	1	
Inputs 10	1520	1	1520	1	
Inputs 2	1520	1	1520	1	
Inputs 3	1520	1	1520	1	
Inputs 4	1520	1	1520	1	
Inputs 5	1520	1	1520	1	
Inputs 6	1520	1	1520	1	
Inputs 7	1520	1	1520	1	
Inputs 8	1520	1	1520	1	
Inputs 9	1520	1	1520	1	

Loop Over Configuration Options During Code Profiling

This example shows how you can use key-value mapping to loop over pairs of configuration parameter options.

Open the model `SILTopModel`.

```
openExample('ecoder/SILPILVerificationExample', ...
    supportingFile='SILTopModel.slx')
model = bdroot;
```

Use key-value mapping to specify loop-over values for the `BuildConfiguration` configuration parameter.

```
params = containers.Map('KeyType','char','ValueType','any');
params("BuildConfiguration") = {"Faster Builds", "Faster Runs"};
```

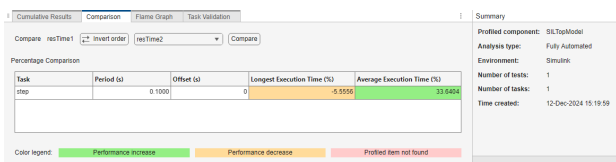
Run SIL tests with execution-time profiling enabled.

```
[resTime1, resTime2] = coder.profile.test.runTests(model, ...
    OverrideSettings=params);
```

To compare execution-time results, use the Code Profile Analyzer app. Display the first set of results.

```
coder.profile.show(resTime1)
```

On the **Time Profiling** tab, select the **Comparison** tab. From the **Select a profile variable** list, choose `resTime2`, which contains the second set of results. Then, click **Compare**.



The values in the **Longest Execution Time (%)** and **Average Execution Time (%)** cells indicate that, when `BuildConfiguration` is set to "Faster Runs", the longest step execution time is slightly longer but the average step execution time is significantly reduced.

Specify loop-over values for an additional configuration parameter.

```
params("MaxStackSize") = {"64", "256"};
```

Run SIL tests with stack usage profiling enabled.

```
[resStack1, resStack2] = coder.profile.test.runTests(model, ...
    CodeMetric="stack", ...
    OverrideSettings=params);
```

Display the first set of results.

```
coder.profile.show(resStack1)
```

On the **Memory Profiling** tab, select the **Comparison** tab. From the **Select a profile variable** list, choose `resStack2`, which contains the second set of results. Then, click **Compare**.

Task	Period (s)	Offset (s)	Highest Stack Usage (bytes) (%)	Average Stack Usage (bytes) (%)
step	0.1000	0	0	0

Color legend: Performance increase (green), Performance decrease (orange), Profiled item not found (red).

The display shows values of zero in the **Highest Stack Usage (bytes) (%)** and **Average Stack Usage (bytes) (%)** cells. In this example, changing the configuration parameter combination from BuildConfiguration="Faster Builds" and MaxStackSize="64" to BuildConfiguration="Faster Runs" and MaxStackSize="256" does not affect stack usage metrics for the generated code.

Customize Automated Code Profiling Using Hooks

This example shows how you can use a `coder.profile.test.TestHook` object to customize automated execution-time profiling.

Open the model `SILTopModel`.

```
openExample('ecoder/SILPILVerificationExample', ...
    supportingFile='SILTopModel.slx')
model = bdroot;
```

In your working folder, provide MATLAB code that specifies a subclass of `coder.profile.test.TestHook` and defines functionality for hooks. For example, `myTestHook.m`.

```
classdef myTestHook < coder.profile.test.TestHook
    methods(Access = public)
        function TestSetup(this)
            disp("-----> PRETESTS_START_HOOK");
        end
        function TestStart(this, modelName, isTaskProfilingStage)
            disp("-----> PRETEST_RUN_HOOK");
        end
        function TestEnd(this, modelName, isTaskProfilingStage)
            disp("-----> POSTTEST_RUN_HOOK");
        end
        function TestCleanup(this, aggregatedResults)
            disp("-----> POSTTESTS_PREEXIT_HOOK");
            aggregatedResults.addDetails("Item1", "Description for Item1");
            aggregatedResults.addDetails("Item2", "Description for Item2");
        end
    end
end
```

Run `coder.profile.test.runTests`, which invokes the test hooks.

```
resTime = coder.profile.test.runTests(model, ...
    "TestHooks", myTestHook());
```

```
-----> PRETESTS_START_HOOK
-----> PRETEST_RUN_HOOK
### Searching for referenced models in model 'SILTopModel'.
### Total of 1 models to build.
### Starting build procedure for: SILTopModel
### Successful completion of build procedure for: SILTopModel
```

Build Summary

Top model targets:

Model	Build Reason	Status	Build Duration

```
=====
SILTopModel  Generated code was out of date.  Code generated and compiled.  0h 0m 9.7842s

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.145s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2022 (C)'.
MEX completed successfully.
### Starting SIL simulation for component: SILTopModel
### Application stopped
### Stopping SIL simulation for component: SILTopModel
-----> POSTTEST_RUN_HOOK
-----> PRETEST_RUN_HOOK
### Searching for referenced models in model 'SILTopModel'.
### Total of 1 models to build.
### Starting build procedure for: SILTopModel
### Successful completion of build procedure for: SILTopModel

Build Summary

Top model targets:

Model          Build Reason          Status          Build Duration
=====
SILTopModel  Generated code was out of date.  Code generated and compiled.  0h 0m 10.157s

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 10.448s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2022 (C)'.
MEX completed successfully.
### Starting SIL simulation for component: SILTopModel
### Application stopped
### Stopping SIL simulation for component: SILTopModel
-----> POSTTEST_RUN_HOOK
-----> POSTTESTS_PREEXIT_HOOK
```

The build output contains hook messages.

To view results in the Code Profile Analyzer app, run:

```
coder.profile.show(resTime)
```

On the **Cumulative Results** tab, the **Test Execution Information** section displays the information that you provided through the `addDetails` function.

Test Execution Information	
Test Note	Full Description
Item1	Description for Item1
Item2	Description for Item2

In the **Results** section of the toolstrip, click **Create PDF**. The PDF report displays the information that you supplied through the `addDetails` function.

Profiled component:	SILTopModel
Environment:	Simulink
Number of tests:	1
Number of tasks:	1
Time unit:	ns
Timer Ticks per Second (MHz):	3595
Item1	Description for Item1
Item2	Description for Item2

Input Arguments

testComponent — Model or function name

character vector | string

Name of Simulink model or MATLAB function that produces generated code. For a MATLAB function, you must also specify the `MATLABConfig` and `MATLABInputs` name-value arguments.

Example: `runTests('myModel');`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `runTests('myModel', Testfile="myTestFile");`

Testfile — Test file

character vector | string

Simulink Test file for model.

Example: `runTests('myModel', Testfile="myTestFile");`

CodeMetric — Metric type

"time" (default) | "stack"

Type of code profiling metric to analyze for generated code:

- "time" — Execution-time metrics
- "stack" — Stack usage metrics

Example: `runTests('myModel', CodeMetric="stack");`

OverrideSettings — Override settings

containers.Map object

Investigate the effect of specific parameters on code execution-time and stack usage. Using a `Map` object, specify::

- As keys, model configuration parameters that you want to override
- For each key, a two-element cell array that contains two parameter values

The function runs two SIL or PIL tests using the `Map` object. For the first test, the function uses the first value of each key. For the second test, the function uses the second value of each key. The function returns two sets of results.

Example: `runTests('myModel', OverrideSettings=containers.Map("MaxStackSize", {"64", "256"}));`

MATLABConfig — MATLAB configuration object

`coder.EmbeddedCodeConfig` object

MATLAB configuration object to use for code generation during SIL or PIL test.

Example: `runTests("myFunction.m", MATLABConfig=myConfigObj, MATLABInputs={ {25, 'Alice', 170}, {30, 'Bob', 180}, {22, 'Charlie', 165}});`

MATLABInputs — Input source for MATLAB function

array of cell arrays

Source of input sets for SIL or PIL testing of MATLAB function, specified as an array of cell arrays. Each input set must be an array of inputs for a single run of the generated function code.

Example: `runTests("myFunction.m", MATLABConfig=myConfigObj, MATLABInputs={ {input1_run1, input2_run1, ...}, {input1_run2, input2_run2, ...}, ... });`

NumIterations — Number of iterations

1 (default) | integer scalar

For advanced target applications, increasing the number of SIL or PIL tests can yield more stable execution times. If you have a Simulink model or MATLAB function that you consider a benchmark, use this argument to specify the number of times the function runs the SIL or PIL test.

Applies only if `CodeMetric` is "time" and `Testfile` is empty.

Example:

`runTests("myFunction.m", MATLABConfig=cfg, MATLABInputs=in, NumIterations=10);`

AppendResultsTo — Results object

`coder.profile.ExecutionTimeSet` object | `coder.profile.ExecutionStackSet` object

Object that analysis adds results to. If you do not specify an object, the function creates a new object.

Example: `runTests('myModel', AppendResultsTo=myNameObject);`

NumTestsForDetailedProfiling — Tests to rerun

'all' (default) | double scalar

Number of most demanding tests to rerun for each task with detailed profiling enabled. Applies to execution-time analysis only.

Example: `runTests('myModel', NumTestsForDetailedProfiling=6);`

TestHooks — Test hooks

`coder.profile.test.TestHook` object

Specify a custom object that inherits from `coder.profile.test.TestHook`. The function invokes object methods at predefined moments during the SIL or PIL test.

Example: `runTests('myModel', TestHooks=myTestHook());`

Verbose — Verbosity

false (default) | true

Verbosity for debugging:

- `false` — No messages shown during analysis.
- `true` — Diagnostic message shown after each step.

Example: `runTests('myModel', Verbose=true);`

Output Arguments

resultsSet — Results set

`coder.profile.ExecutionTimeSet` object | `coder.profile.ExecutionStackSet` object

Object that stores test results returned by SIL or PIL tests. If you specify `OverrideSettings`, object stores results from the first input set. Object type is:

- `coder.profile.ExecutionTimeSet` if `CodeMetric` is "time"
- `coder.profile.ExecutionStackSet` if `CodeMetric` is "stack"

secondResultsSet — Second results set

`coder.profile.ExecutionTimeSet` object | `coder.profile.ExecutionStackSet` object

If you specify `OverrideSettings`, object stores results from the second input set. Object type is:

- `coder.profile.ExecutionTimeSet` if `CodeMetric` is "time"
- `coder.profile.ExecutionStackSet` if `CodeMetric` is "stack"

If you do not specify `OverrideSettings`, object is empty.

Version History

Introduced in R2024a

R2025a: Enhancements for Automated Code Execution-Time and Stack Usage Profiling

To enhance automated code execution-time and stack usage profiling workflows, the function supports these name-value arguments:

- `OverrideSettings`
- `MATLABConfig`
- `MATLABInputs`
- `NumIterations`
- `TestHooks`

The function can also return a second output argument `secondResultsSet`.

See Also

`coder.profile.test.generateTests` | `coder.profile.ExecutionTimeSet` | `coder.profile.ExecutionStackSet` | `coder.profile.test.TestHook`

Topics

"Aggregate Execution-Time Profiles to Identify Worst Execution"

"Aggregate Stack Usage Profiles to Identify Most Demanding Execution"
"Analyze Execution Times of Critical Paths"
"Overview of Signal Loading Techniques"
"Load Input Data for Basic Test Cases"

coder.profile.test.TestHook

Define custom callback functions for `coder.profile.test.runTests`

Description

Use a `coder.profile.test.TestHook` object to define custom callback functions or hooks for the `coder.profile.test.runTests` function. When you use `coder.profile.test.runTests` to run software-in-the-loop (SIL) or processor-in-the-loop (PIL) tests of a model, the function invokes the hooks at different execution points.

Creation

Create a subclass of `coder.profile.test.TestHook` that defines functionality for hooks `TestSetup`, `TestStart`, `TestEnd`, and `TestCleanup`. The `coder.profile.test.runTests` function uses your `coder.profile.test.TestHook` object to invoke:

- `TestSetup` at the start of the function, before running any SIL or PIL tests.
- `TestStart` just before running each SIL or PIL test.
- `TestEnd` just after running each SIL or PIL test.
- `TestCleanup` just before the end of the function, after running all SIL or PIL tests.

Examples

Implement Test Hooks for `coder.profile.test.runTests`

For an example that shows you how to implement test hooks for the `coder.profile.test.runTests` function, see “Customize Automated Code Profiling Using Hooks” on page 1-324.

Version History

Introduced in R2025a

See Also

`coder.profile.test.runTests` | `coder.profile.ExecutionTimeSet` | `coder.profile.ExecutionStackSet`

Topics

“Customize Automated Code Profiling Using Hooks” on page 1-324

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 MATLAB setting.

`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.

Version History

Introduced in R2017b

See Also

[sharedCodeUpdate](#) | [crossReleaseImport](#)

Topics

“Cross-Release Shared Utility Code Reuse”

“Cross-Release Code Integration”

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)
sharedCodeUpdate(protectedModel, destinationFolder)
```

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.

`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.

`sharedCodeUpdate(protectedModel, destinationFolder)` copies shared utility files for the protected model to the shared code folder.

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, 'EarlierWork', '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, 'EarlierWork', '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 “Code generation folder” 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.

protectedModel — Protected model file

character vector

File path for protected model. File name of protected model must have .slxp extension.

Version History

Introduced in R2016b

See Also

crossReleaseImport

Topics

“Cross-Release Shared Utility Code Reuse”

“Cross-Release Code Integration”

“Reference Protected Models from Third Parties”

getStatus

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

[addInputData](#) | [run](#) | [addBaseline](#)

Topics

[“Verify Numerical Equivalence with CGV”](#)

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' | 'void'

Built-in data type or fixed-point data type to use for the code replacement argument. You can specify fixed-point data types using the `fixdt` function from Fixed-Point Designer™ software; for example, `'fixdt(1,16,2)'`. To specify a pointer argument, use the symbol `*` after the data type name; for example, `'double*'`.

Example: 'integer'

Example: 'double*'

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.

Version History

Introduced in R2008a

See Also

`addConceptualArg`

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

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: 'dl'

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.

Version History

Introduced in R2013a

See Also

addDWorkArg

Topics

“Define Code Replacement Library Optimizations”

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.

Note The code generator enables you to extend the range of supported hardware by using the `target.create` and `target.add` functions to register new devices. After you register a new device, you can create a `coder.Hardware` object for this new device as well.

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.

Version History

Introduced in R2015b

See Also

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.HardwareImplementation` |
`codegen`

RTW.TfBlasEntryGenerator

Create code replacement table entry for a BLAS operation

Syntax

```
obj = RTW.TfBlasEntryGenerator
```

Description

`obj = RTW.TfBlasEntryGenerator` 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.TfTable;

arch = computer('arch');
compilerName = 'microsoft';
LibPath = fullfile('${MATLAB_ROOT}', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.TfBlasEntryGenerator;

libExt = 'lib';

setTfLCOperationEntryParameters(op_entry, ...
    'Key', 'RTW_OP_MUL', ...
    'Priority', 100, ...
    'ImplementationName', 'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32_crl.h', ...
    'ImplementationHeaderPath', fullfile('${MATLAB_ROOT}', 'extern', 'include'), ...
    'AdditionalLinkObjs', {'libmwbblascompat32.' 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.

Version History

Introduced in R2010a

See Also

`RTW.TflCBlasEntryGenerator` | `RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"

"Matrix Multiplication Operation to MathWorks BLAS Code Replacement"

"Code You Can Replace from MATLAB Code"

"Code You Can Replace From Simulink Models"

RTW.TflBlockEntry Class

Namespace: RTW

Create block replacement entry

Description

Use the `RTW.TflBlockEntry` class to finely control the specification of a block replacement entry. For block replacement entries, you can specify match criteria such as block parameter settings that the entry must match for code replacement. You can specify implementation functions for each model system function that the block generates, such as initialize, update, and terminate functions.

The `RTW.TflBlockEntry` class is a `handle` class.

Creation

Description

`entry = RTW.TflBlockEntry` creates an `RTW.TflBlockEntry` object.

Properties

Key — Type of block to replace

`'DiscreteFIR' | 'Biquad Filter' | 'FFT' | 'IFFT' | 'FIR Decimation' | 'FIR Interpolation'`

Type of block to replace, specified as a character array.

Example: `'DiscreteFIR'`

AdditionalHeaderFiles — Additional header files for block replacement entry

`{}` (default) | cell array of character vectors

Additional header files for a block replacement entry, specified as a cell array of character vectors.

Example: `'AdditionalHeaderFiles',{}`

BlockParams — Block dialog properties to match

`[]` (default) | array of block property handles

Block dialog properties for the block replacement entry to match, specified as an array of block property handles, such as the handle created by the `RTW.BlockProperty` function. Each block property handle specifies the name and value of a block property that the block replacement entry matches during code replacement. You can add a block property by using the `addBlockProperty` method.

Example: `RTW.BlockProperty('FilterStructure','Direct form')`

BlockParamArgs — Numeric block parameters used by implementation functions

`[]` (default) | array of argument handles

Numeric block parameters whose values are used by the implementation functions, specified as an array of argument handles. These parameters are called *block parameter arguments*. Implementation functions can use block parameter argument data directly (by referencing the parameter) or indirectly (by referencing a derived parameter that uses the block parameter argument).

To use a block parameter value in an implementation function, you must specify the data type information of the parameter by adding a block parameter argument as an argument handle in `BlockParamArgs`. Create an argument handle that specifies the data type by using the `getTflArgFromString` function or the `RTW.TflArgMatrix` or `RTW.TflArgNumeric` classes. Add the argument by using the `addBlockParamArg` method.

During replacement, the code generator uses the data type information to:

- Match blocks that use the corresponding data type for the parameter.
- Construct the arguments for calling the implementation code that uses the block parameter argument.

Example: `[RTW.TflArgMatrix('Coefficient','RTW_IO_INPUT','int16')]`

DerivedBlockParams — Derived parameters used for implementation function arguments

`[]` (default) | cell array of character vectors

Derived parameters that one or more implementation functions use for arguments, specified as a cell array of character vector MATLAB expressions. The expressions must follow the format `'derivedParameter = expression'` where:

- The right side of the equation is a constant, conceptual argument, block parameter, or a MATLAB function that is on the path and supports code generation.
- If the right side of the equation uses variables such as block parameters, conceptual arguments, or other derived block parameters, each variable follows the format `<%VarName>`.
- The left side of the expression is the derived parameter that an implementation function of the block replacement entry uses.

Depending on the expression, the derived parameter is generated as a variable in the generated code or is evaluated and inlined as a constant in the code. Derived parameters that use the functions `size`, `length`, and `numel` are evaluated during code generation and inlined as constants. A derived parameter does not appear in the generated code if none of the block replacement entry implementation functions use the parameter.

Example: `[{'coeffDim = size(<%Coefficients>)'}, {'dim1 = <%coeffDim>(1)'}, {'dim2 = <%coeffDim>(2)'}]`

Example: `[{'InputDim = size(<%u1>)'}]`

Example: `'tempBuffer = <%u1>'`

GenCallback — Callback that follows code generation

`''` (default) | `'RTW.copyFileToBuildDir'`

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.

Example: `'GenCallback','RTW.copyFileToBuildDir'`

ImplementationVector — Implementation functions

[] (default) | cell array

Implementation functions that replace the generated code for the block, specified as an N -by-2 cell array in which the first column contains the character vector name of a model system function and the second column contains an `RTW.CImplementation` object that represents the implementation function.

Model system functions include:

- 'initialize'
- 'update'
- 'output'
- 'terminate'

You can add an implementation function by using the `addImplementation` method.

Example: `['initialize'];{initImplementationObj}]`

Priority — Search priority for block replacement entry

100 (default) | integer from 0 to 100

Search priority for the block replacement entry relative to other entries for the block, specified as an integer from 0 to 100. The highest priority is 0 and the lowest priority is 100. If the code replacement table provides two entries of the same block key and conceptual argument list, the entry with the higher priority is used for replacement.

Example: 100

SideEffects — Option to optimize eliminate the implementation function for optimization

false (default) | true

Option to instruct the code generator not to attempt to optimize away the implementation function described by this entry, specified as `false` or `true`. 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`.

Methods**Public Methods**

<code>addBlockProperty</code>	Add block property to a block replacement entry
<code>addBlockParamArg</code>	Add block property data type information
<code>addImplementation</code>	Add implementation function to block replacement entry
<code>createDWorkArg</code>	Create DWork argument for a block replacement entry

Examples**Create Block Replacement Entry**

Create a block replacement entry for the Discrete FIR Filter block.

Create the block replacement entry by using the key `DiscreteFir` for the Discrete FIR Filter block. Set the priority of the entry to 1.

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

```
entry = RTW.TflBlockEntry;  
entry.Key = 'DiscreteFir';  
entry.Priority = 1;
```

Specify the block properties to match by using the `RTW.BlockProperty` function. Specify that the block entry matches Discrete FIR Filter blocks that have `FilterStructure` set to `Direct form` and `InputProcessing` set to `Columns as channels (frame based)`.

```
prop1 = RTW.BlockProperty('FilterStructure', 'Direct form');  
addBlockProperty(entry, prop1);  
prop2 = RTW.BlockProperty('InputProcessing', 'Columns as channels (frame based)');  
addBlockProperty(entry, prop2);
```

Create the conceptual representation of the block by specifying block inputs and outputs. Add the first output with the type `single` to the entry.

```
arg = getTflArgFromString(hLib, 'y1', 'single');  
addConceptualArg(entry, arg);
```

Add the specifications for the implementation functions to the entry. Add the initialization function `init_impl`, which uses the derived block parameter `numTaps` as an input argument. Because the derived parameter uses the block parameter `Coefficients`, specify the data type for `Coefficients` by adding a block parameter argument.

```
impl = RTW.CImplementation;  
impl.Name = 'init_impl';  
  
blockParamArg = RTW.TflArgMatrix('Coefficients', 'RTW_I0_INTPUT', 'int16');  
addBlockParamArg(entry, blockParamArg);  
  
entry.DerivedBlockParams{1} = 'numTaps = length(<%Coefficients>)';  
arg = getTflArgFromString(hLib, 'numTaps', 'uint16');  
addArgument(impl, arg);  
  
addImplementation(entry, 'initialize', impl);
```

Add the block replacement entry to the code replacement table.

```
addEntry(hLib, entry);
```

Version History

Introduced in R2024a

See Also

`addEntry` | `addDWorkArg`

Topics

“Replace Code Generated from Discrete FIR Filter Blocks”

addBlockParamArg

Class: RTW.TflBlockEntry

Namespace: RTW

Add block property data type information

Syntax

```
addBlockParamArg(blockEntry,arg)
```

Description

`addBlockParamArg(blockEntry,arg)` specifies the block property and data type information, represented by `arg`, that the block replacement entry `blockEntry` must match. You must add data type information for the block properties that you use for derived arguments or implementation arguments in the block entry.

Input Arguments

blockEntry — Block entry object

RTW.TflBlockEntry object

Block entry object, specified as an RTW.TflBlockEntry object.

arg — Block property data type representation

argument handle

Block property data type representation, specified as an argument handle. Create an argument handle that specifies the property and data type by using the `getTflArgFromString` function or the `RTW.TflArgMatrix` or `RTW.TflArgNumeric` class. When you create the argument handle, specify the block property name as the argument name.

Example: `RTW.TflArgMatrix('Coefficient','RTW_IO_INPUT','int16')`

Examples

Add Derived Parameter Using Length of Coefficients Block Property

Create an entry for the Discrete FIR filter in which the implementation function uses the length of the block property `Coefficients`.

Create a block replacement entry object and specify the key for the Discrete FIR filter block.

```
entry = RTW.TflBlockEntry;
entry.Key = 'DiscreteFir';
```

Create an argument that represents the block property and the data type that the implementation requires. For this example, specify that the `Coefficients` property must be set to a value that is a matrix of type `int16`.

```
arg = RTW.TflArgMatrix('Coefficients','RTW_IO_INPUT','int16');
```

During code replacement, the entry will match blocks in which the value of the `Coefficients` property is a matrix of type `int16`.

Specify the dimension range of the `Coefficients` matrix.

```
arg.DimRange = [1 1; Inf Inf];
```

Add the property information to the block entry.

```
addBlockParamArg(entry,arg);
```

To check the property information in the entry, use the `BlockParamArgs` property.

```
entry.BlockParamArgs
```

```
ans =
```

```
    TflArgMatrix with properties:
```

```
        BaseTypeIsUnsizeInt: 0
        BaseTypeIsUnsizeLong: 0
        BaseTypeIsUnsizeLongLong: 0
            DimRange: [2×2 double]
        CheckSlope: 1
        CheckBias: 1
        CheckType: 1
        MetaData: []
            Name: 'Coefficient'
            Type: [1×1 coder.descriptor.types.Matrix]
            IOType: 'RTW_IO_INPUT'
        Descriptor: []
```

Version History

Introduced in R2024a

See Also

[RTW.TflBlockEntry](#) | [getTflArgFromString](#) | [RTW.TflArgMatrix](#) | [RTW.TflArgNumeric](#)

addBlockProperty

Class: RTW.TflBlockEntry

Namespace: RTW

Add block property to a block replacement entry

Syntax

```
RTW.TflBlockEntry.addBlockProperty(blockEntry,blockProperty)
```

Description

RTW.TflBlockEntry.addBlockProperty(blockEntry,blockProperty) adds the block property blockProperty to the block replacement entry object blockEntry. The entry replaces only blocks that use the specified value for the block property.

Input Arguments

blockEntry — Block entry object

RTW.TflBlockEntry object

Block entry object, specified as an RTW.TflBlockEntry object.

Example: RTW.TflBlockEntry

blockProperty — Block property

block property handle

Block property, specified as a block property handle created by the RTW.BlockProperty function. During code replacement, the block entry matches blocks that use the property settings specified by the block properties that you add to the entry.

Example: RTW.BlockProperty('FilterStructure', 'Discrete')

Examples

Replace Filter Block That Uses Direct Form Structure and Frame-based Processing

Create a block entry for the Discrete FIR Filter block that matches blocks that use the Direct form filter structure and frame-based input processing.

```
entry = RTW.TflBlockEntry;
entry.Key = 'DiscreteFir';

prop1 = RTW.BlockProperty('FilterStructure', 'Direct form');
addBlockProperty(entry, prop1);
prop2 = RTW.BlockProperty('InputProcessing', 'Columns as channels (frame based)');
addBlockProperty(entry, prop2);
```

During code replacement, the entry for the Discrete FIR Filter block matches blocks that have `FilterStructure` set to `Direct` form and `InputProcessing` set to `Columns as channels` (frame based).

Version History

Introduced in R2024a

See Also

`RTW.TflBlockEntry`

addImplementation

Class: RTW.TflBlockEntry

Namespace: RTW

Add implementation function to block replacement entry

Syntax

```
RTW.TflBlockEntry.addImplementation(blockEntry,modelFunc,implFunc)
```

Description

`RTW.TflBlockEntry.addImplementation(blockEntry,modelFunc,implFunc)` adds the specified implementation function `implFunc` to the block replacement entry object `blockEntry`. When code replacement occurs for the block replacement entry, the implementation function replaces the model system function `modelFunc` that you specify, such as the initialize, update, or terminate function.

Input Arguments

blockEntry — Block entry object

RTW.TflBlockEntry object

Block entry object, specified as an RTW.TflBlockEntry object.

modelFunc — Model system function

'initialize' | 'update' | 'output' | 'terminate'

Model system function for the implementation function to replace, specified as 'initialize', 'update', 'output', or 'terminate'.

Example: 'initialize'

implFunc — Implementation function

RTW.CImplementation object

Implementation function that replaces the model system function, specified as an RTW.CImplementation object.

Examples

Replace Initialize and Terminate Functions for Filter Block

Create a block replacement entry that uses different implementation functions to replace the initialize and terminate functions for a Discrete FIR Filter block.

```
entry = RTW.TflBlockEntry;
entry.Key = 'DiscreteFir';
```

```
implInit = RTW.CImplementation;  
implInit.Name = 'filterInit';  
  
addImplementation(entry,'initialize', implInit);  
  
implTerm = RTW.CImplementation;  
implTerm.Name = 'filterTerm';  
  
addImplementation(entry,'terminate', implTerm);
```

When the generated code replaces a Discrete FIR Filter block using this entry, the code uses the initialize function `filterInit` and the terminate function `filterTerm`.

Version History

Introduced in R2024a

See Also

`RTW.CImplementation`

createDWorkArg

Class: RTW.TflBlockEntry

Namespace: RTW

Create DWork argument for a block replacement entry

Syntax

```
arg = createDWorkArg(blockEntry,type,Name1=Value1,...,NameN=ValueN)
```

Description

`arg = createDWorkArg(blockEntry,type,Name1=Value1,...,NameN=ValueN)` creates a handle for a DWork argument of type *type* that you can add to a block replacement entry `blockEntry`.

Input Arguments

blockEntry — Block entry object

RTW.TflBlockEntry object

Block entry object, specified as an RTW.TflBlockEntry object.

type — Data type

'Integer' | 'Boolean' | 'Half' | 'Single' | 'Double' | 'Fixed' | 'Complex' | 'Matrix' | 'Pointer' | 'Struct' | character vector | string scalar

Data type of the DWork argument, specified as a character vector or string scalar. You can specify one of the supported type strings or a character vector or string that creates a data type, such as 'fixdt(true,16,0)' or 'int32'.

Example: 'Pointer'

Example: 'fixdt(true,16,0)'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where *Name* is the argument name and *Value* is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Name='crlDWorkArg'`

Name — Argument name

character vector | string scalar

Argument name for the DWork vector, specified as a character vector or string scalar.

Example: `Name='CRLDWork'`

StructData — Structure type information

structure

Structure type information, specified as a structure with these fields:

- **Identifier** — String name of the structure.
- **Elements** — Array of elements in the structure. Each element is a structure with the fields **Type** and **Identifier**.

For example, this code creates the structure type information in the structure `ssTypeP`.

```
se1.Identifier = 'Coeffs';  
se1.Type = 'single*';  
  
se2.Identifier = 'InitStates';  
se2.Type = 'single';  
  
se3.Identifier = 'CoeffsSize';  
se3.Type = 'uint16';  
  
ssTypeP.Identifier = 'InitStruct';  
ssTypeP.Elements = [se1, se2, se3];
```

Type — Data type of the argument

character vector | string scalar

Data type of the argument, specified as a character vector or string scalar. You can specify built-in MATLAB data types such as `uint8`, `boolean`, `double`, and others.

Example: `Type='uint8'`

BaseType — Base data type

character vector | string scalar

Base data type of a matrix, pointer, or complex DWork vector, specified as a character vector or string scalar.

Example: `BaseType='single'`

Output Arguments

arg — Handle to the created DWork argument

handle

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

Version History

Introduced in R2024a

See Also

`RTW.TflBlockEntry`

Topics

“Replace Code Generated from Discrete FIR Filter Blocks”

“Use Derived Data in Replacement Code Implementations”

RTW.CImplementation

Specify implementation function for a block replacement entry

Description

Use RTW.CImplementation objects to specify implementation functions for a block replacement entry. You can specify implementation functions for the initialize, update, output, and terminate functions.

Creation

Syntax

```
implementation = RTW.CImplementation
```

Description

`implementation = RTW.CImplementation` creates an RTW.CImplementation object.

Properties

HeaderFile — Name of header file that declares implementation function

`''` (default) | character vector | string scalar

Name of the header file that declares the implementation function, specified as a character vector or string scalar.

Example: `'ImplementationHeaderFile', ''`

HeaderPath — Path to implementation header file

`''` (default) | character vector | string scalar

Full path to the implementation header file, specified as a character vector or string scalar.

Example: `'ImplementationHeaderPath', ''`

Name — Name of implementation function

`''` (default) | character vector | string scalar

Name of the implementation function, specified as a character vector or string scalar.

Example: `'arm_fir_init'`

SourceFile — Name of implementation source file

`''` (default) | character vector | string scalar

Name of the implementation source file, specified as a character vector or string scalar.

Example: `'ImplementationSourceFile', ''`

SourcePath — Path to implementation source file

`''` (default) | character vector | string scalar

Full path to the implementation source file, specified as a character vector or string scalar.

Example: `'ImplementationSourcePath', ''`

Object Functions

`addArgument` Add argument to `RTW.CImplementation` object

`setReturn` Add return argument to `RTW.CImplementation` object

Examples**Add Initialization Function to a Block Replacement Entry**

Specify an implementation function to replace the initialize function of a Discrete FIR Filter block.

Create a block replacement entry for the Discrete FIR Filter block.

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

```
entry = RTW.TflBlockEntry;  
entry.Key = 'DiscreteFir';  
entry.Priority = 1;
```

Create the `RTW.CImplementation` object that represents the implementation function. Specify the name and header file for the function.

```
impl = RTW.CImplementation;  
impl.Name = 'arm_fir_init';  
impl.HeaderFile = 'arm_math.h';
```

Specify the arguments that implementation function takes.

```
arg = getTflArgFromString(hLib, 'u1', 'uint16');  
addArgument(impl, arg);
```

Add the implementation function to the block replacement entry as the replacement initialization function.

```
addImplementation(entry, 'initialize', impl);
```

Version History

Introduced in R2024a

See Also

`addImplementation` | `RTW.TflBlockEntry`

Topics

“Replace Code Generated from Discrete FIR Filter Blocks”

addArgument

Namespace: RTW

Add argument to RTW.CImplementation object

Syntax

```
addArgument(implementation, arg)
```

Description

`addArgument(implementation, arg)` adds the argument to the RTW.CImplementation object that represents the implementation function specification `implementation`.

Examples

Specify an Input Argument for an Implementation Function

Add an input argument to the specification for an implementation function. For this example, the implementation function replaces the initialize function of a Discrete FIR Filter block.

Create a block replacement entry for the Discrete FIR Filter block.

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

```
entry = RTW.TflBlockEntry;  
entry.Key = 'DiscreteFir';
```

Create the RTW.CImplementation object that represents the implementation function. Specify the name and header file for the function.

```
impl = RTW.CImplementation;  
impl.Name = 'arm_fir_init';  
impl.HeaderFile = 'arm_math.h';
```

Add an input argument to the implementation function object.

```
arg = getTflArgFromString(hLib, 'u1', 'uint16');  
addArgument(impl, arg);
```

Add the implementation function to the block replacement entry as the replacement initialization function.

```
addImplementation(entry, 'initialize', impl);
```

Input Arguments

implementation — Implementation function

RTW.CImplementation object

Implementation function, specified as an `RTW.CImplementation` object.

arg — Implementation function argument

argument handle

Implementation function argument, specified as a handle such as the argument handle returned by the function `getTflArgFromString`.

Example: `getTflArgFromString('u1','uint16')`

Version History

Introduced in R2024a

See Also

`RTW.CImplementation` | `getTflArgFromString`

setReturn

Namespace: RTW

Add return argument to RTW.CImplementation object

Syntax

```
setReturn(implementation,arg)
```

Description

setReturn(implementation,arg) adds the return argument to the RTW.CImplementation object that represents the implementation function specification `implementation`.

Examples

Specify a Void Return for an Implementation Function

Specify a void return for an implementation function for the Discrete FIR Filter block.

Create a block replacement entry for the Discrete FIR Filter block.

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

```
entry = RTW.TflBlockEntry;
entry.Key = 'DiscreteFir';
```

Create the RTW.CImplementation object that represents the implementation function. Specify the name and header file for the function.

```
impl = RTW.CImplementation;
impl.Name = 'arm_fir_init';
impl.HeaderFile = 'arm_math.h';
```

Add an input argument to the implementation object.

```
arg = getTflArgFromString(hLib, 'u1', 'uint16');
addArgument(impl, arg);
```

Specify a void return for the implementation object.

```
arg = hLib.getTflArgFromString('void', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
setReturn(impl,arg);
```

Add the implementation function to the block replacement entry as the replacement initialization function.

```
addImplementation(entry, 'initialize', impl);
```

Input Arguments

implementation — Implementation function

RTW.CImplementation object

Implementation function, specified as an RTW.CImplementation object.

arg — Implementation function return argument

argument handle

Implementation function return argument, specified as a handle such as the argument handle returned by the function `getTflArgFromString`.

Example: `getTflArgFromString('void','void');`

Version History

Introduced in R2024a

See Also

RTW.CImplementation | `getTflArgFromString`

RTW.BlockProperty

Create a block property specification for a block replacement entry

Syntax

```
prop = RTW.BlockProperty(PropName,PropValue)
```

Description

`prop = RTW.BlockProperty(PropName,PropValue)` creates a block property that has the specified property name and value. Adding the property to an `RTW.TflBlockEntry` object specifies the property setting for the block replacemententry to match during code replacement.

Examples

Replace Code from Direct Form Filter Blocks

This example shows how to create a block replacement entry that replaces code from Discrete FIR Filter blocks that use the direct form filter structure.

Create a block replacement entry for the Discrete FIR Filter block.

```
entry = RTW.TflBlockEntry;  
entry.Key = 'DiscreteFir';
```

Create a block property that specifies the property `FilterStructure` and the value `Direct form`.

```
prop = RTW.BlockProperty('FilterStructure', 'Direct form');
```

Add the block property to the block replacement entry.

```
addBlockProperty(entry,prop);
```

When you use the block replacement entry in a code replacement table, the entry matches Discrete FIR Filter blocks that have `FilterStructure` set to `Direct form`.

Input Arguments

PropName — Block property name

character vector | string scalar

Block property name to match during code replacement, specified as a character vector or string scalar. The property can be a dialog property or an algorithm property of the block. To find the available dialog properties of a block, click the block in the model and enter `get_param(gcf, 'DialogParameters')` at the command line. To find the available algorithm properties of a block, click the block in the model and enter `get_param(gcf, 'AlgorithmParameters')` at the command line.

Example: `'FilterStructure'`

PropValue — Property value to match

character vector | string scalar

Property value to match during code replacement, specified as a character vector or string scalar. To find the value of a property for a block, click the block in the model and enter `get_param(gcb, PropertyName)` at the command line.

Example: 'Direct form'

Output Arguments**prop — Handle to the created block property**

handle

Handle to the created block property. Add the block property to a block replacement entry by specifying the handle as an input to the `addBlockProperty` method.

Version History

Introduced in R2024a

See Also

`addBlockProperty` | `RTW.TflBlockEntry`

RTW.Tf1CB1asEntryGenerator

Create code replacement table entry for a CBLAS operation

Syntax

`obj = RTW.Tf1CB1asEntryGenerator`

Description

`obj = RTW.Tf1CB1asEntryGenerator` 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.Tf1Table;

arch = computer('arch');
compilerName = 'my_compiler';
LibPath = fullfile('${MATLAB_ROOT}', 'extern', ...
    'lib', arch, compilerName);

op_entry = RTW.Tf1CB1asEntryGenerator;

libExt = 'lib';

setTf1C0perationEntryParameters(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.

Version History

Introduced in R2010a

See Also

`RTW.TflBlasEntryGenerator` | `RTW.TflCOperationEntry` | `RTW.TflTable`

Topics

"Define Code Replacement Library Optimizations"

"Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement"

"Code You Can Replace from MATLAB Code"

"Code You Can Replace From Simulink Models"

RTW.TflCFunctionEntry

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.

Version History

Introduced in R2007b

See Also

`RTW.TflCFunctionEntryML` | `RTW.TflTable`

Topics

- “Define Code Replacement Library Optimizations”
- “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”

RTW.TflCFunctionEntryML Class

Namespace: RTW

Custom code replacement table function entry

Description

Represent a custom code replacement function entry. Derive a subclass from RTW.TflCFunctionEntryML and, in the subclass, implement a `do_match` method that specifies the match criteria for the code replacement entry.

```
classdef TflCustomFunctionEntry < RTW.TflCFunctionEntryML
    methods
        function ent = do_match(hThis, ...
            hCS0, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong, ... %#ok
            targetBitPerLongLong) %#ok

            % Implement custom do_match function

        end
    end
end
end
```

Creation

Description

`classdef TflCustomFunctionEntry < RTW.TflCFunctionEntryML` creates a subclass for creating custom function entry objects.

Version History

Introduced in R2010a

See Also

RTW.TflCFunctionEntry

Topics

“Customize Match and Replacement Process”

RTW.TfIcOperationEntry

Create code replacement table entry for an operator

Syntax

```
obj = RTW.TfIcOperationEntry
```

Description

`obj = RTW.TfIcOperationEntry` 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.TfIcOperationEntry;
```

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.

Version History

Introduced in R2007b

See Also

`RTW.TfIcOperationEntryGenerator` | `RTW.TfIcOperationEntryGenerator_NetSlope` | `RTW.TfIcOperationEntryML` | `RTW.TfIcTable`

Topics

- “Define Code Replacement Library Optimizations”
- “Scalar Operator Code Replacement”
- “Replace Addition and Subtraction Operator Code”
- “Small Matrix Operation to Processor Code Replacement”
- “Code You Can Replace from MATLAB Code”
- “Code You Can Replace From Simulink Models”

RTW.TfIcOperationEntryML Class

Namespace: RTW

Custom code replacement table operation entry

Description

Represent a custom code replacement operation entry. Derive a subclass from RTW.TfIcOperationEntryML and, in the subclass, implement a `do_match` method that specifies the match criteria for the code replacement entry.

```
classdef MyElemMultEntry < RTW.TfIcOperationEntryML
    methods
        function ent = do_match(hThis, ...
            hCS0, ... %#ok
            targetBitPerChar, ... %#ok
            targetBitPerShort, ... %#ok
            targetBitPerInt, ... %#ok
            targetBitPerLong, ... %#ok
            targetBitPerLongLong ) %#ok

            % Implement custom do_match function

        end
    end
end
```

Creation

Description

`classdef MyElemMultEntry < RTW.TfIcOperationEntryML` creates a subclass for creating operation entry objects.

Version History

Introduced in R2010a

See Also

RTW.TfIcOperationEntry

Topics

“Customize Match and Replacement Process”

RTW.TflCOperationEntryGenerator

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.

Version History

Introduced in R2008a

See Also

`RTW.TflCOperationEntry` | `RTW.TflCOperationEntryGenerator_NetSlope` | `RTW.TflCOperationEntryML` | `RTW.TflTable`

Topics

“Define Code Replacement Library Optimizations”
 “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”

RTW.TfIcOperationEntryGenerator_NetSlope

Create code replacement table entry for a net slope fixed-point operation

Syntax

```
obj = RTW.TfIcOperationEntryGenerator_NetSlope
```

Description

`obj = RTW.TfIcOperationEntryGenerator_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.TfIcOperationEntryGenerator_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.

Version History

Introduced in R2008b

See Also

`RTW.TfIcOperationEntry` | `RTW.TfIcOperationEntryGenerator` | `RTW.TfIcOperationEntryML`

Topics

- "Define Code Replacement Library Optimizations"
- "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"

RTW.TfLCSemaphoreEntry

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.

Version History

Introduced in R2010a

See Also

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

RTW.TflTable

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.

Version History

Introduced in R2007b

See Also

`addEntry` | `createCRLEntry`

Topics

“Define Code Replacement Library Optimizations”

“Code You Can Replace from MATLAB Code”

“Code You Can Replace From Simulink Models”

RTW.TflArgNumeric Class

Namespace: RTW

Numeric argument for code replacement entry

Description

Use the `RTW.TflArgNumeric` class to finely control the specification of a numeric argument for a code replacement entry.

Creation

Description

`RTW.TflArgNumeric(name, IOType)` creates a numeric argument, specifies the name, and specifies whether the argument is an input or an output.

Input Arguments

name — Argument name

character vector | string scalar

Argument name, specified as a character vector or string scalar.

IOType — Input or output

'RTW_IO_INPUT' (default) | 'RTW_IO_OUTPUT'

Input or output argument, specified as either 'RTW_IO_INPUT' or 'RTW_IO_OUTPUT'.

Properties

Name — Argument name

character vector | string scalar

Name of the argument, specified as a character vector or string scalar.

Example: 'u1'

IOType — Input or output

'RTW_IO_INPUT' (default) | 'RTW_IO_OUTPUT'

Input or output argument, specified as either 'RTW_IO_INPUT' or 'RTW_IO_OUTPUT'.

CheckSlope — Match slope value

true or 1 (default) | false or 0

Whether to match the slope value of the argument of a fixed-point operator, specified as true (1) or false (0).

CheckBias — Match bias value

true or 1 (default) | false or 0

Whether to match the bias value of the argument of a fixed-point operator, specified as `true` (1) or `false` (0).

Examples

Create a Numeric Argument in a Code Replacement Entry

In this example, you create a code replacement entry for a fixed-point operation that has a numeric output argument. When you create a code replacement library, include this code in the function that defines the code replacement table.

Create an entry object.

```
% Create a table object
hTable = RTW.TflTable;

% Create an entry
hEntry = RTW.TflCOperationEntryML;
```

Create the conceptual representation of the argument by creating an `RTW.TflArgNumeric` object. Use the object to specify properties of the argument. For example, set the argument as an output.

```
arg = getTflArgFromString(hTable, 'u1', 'int32');
arg.IOType = 'RTW_IO_OUTPUT';
```

Add the conceptual argument to the entry.

```
hEntry.addConceptualArg(arg);
```

After you create the entry, you can set the entry parameters, add other arguments, and add the entry to the code replacement table.

Version History

Introduced in R2007b

See Also

`addConceptualArg` | `RTW.TflCFunctionEntry` | `RTW.TflCOperationEntry`

Topics

“Slope Bias Scaling Code Replacement”

“Math Function Code Replacement”

RTW.TflArgMatrix Class

Namespace: RTW

Matrix argument for code replacement entry

Description

Use the `RTW.TflArgMatrix` class to finely control the specification of a matrix argument for a code replacement entry.

Creation

Description

`RTW.TflArgMatrix(name, IOType)` creates a matrix argument, specifies the name, and specifies whether the argument is an input or an output.

Input Arguments

name — Argument name

character vector | string scalar

Argument name, specified as a character vector or string scalar.

IOType — Input or output

'RTW_IO_INPUT' (default) | 'RTW_IO_OUTPUT'

Input or output argument, specified as either 'RTW_IO_INPUT' or 'RTW_IO_OUTPUT'.

Properties

Name — Argument name

character vector | string scalar

Name of the argument, specified as a character vector or string scalar.

Example: 'u1'

IOType — Input or output

'RTW_IO_INPUT' (default) | 'RTW_IO_OUTPUT'

Input or output argument, specified as either 'RTW_IO_INPUT' or 'RTW_IO_OUTPUT'.

CheckSlope — Match slope value

true or 1 (default) | false or 0

Whether to match the slope value of the base type of the argument for a fixed-point operator, specified as true (1) or false (0).

CheckBias — Match bias value

true or 1 (default) | false or 0

Whether to match the bias value of the base type of the argument for a fixed-point operator, specified as true (1) or false (0).

DimRange — Range of matrix dimensions

[1; Inf] (default) | 2-by-N matrix

Range of matrix dimensions, specified as a 2-by-N matrix, where N is the number of dimensions of the matrix argument. Specify the dimension range in the format [Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]. The first row specifies the minimum value for each dimension and the second row specifies the maximum value for each dimension. For example, [2 2; Inf Inf] means a two-dimensional matrix of size 2-by-2 or larger.

Example: [2 2; Inf Inf]

Examples

Create a Matrix Argument in a Code Replacement Entry

In this example, you create a code replacement entry for a function that takes a matrix as an input argument. When you create a code replacement library, include this code in the function that defines the code replacement table.

Create an entry object and set the parameters for the entry.

```
% Create an entry
hEntry = RTW.TfLCOperationEntry;

% Create entry parameters
hEntry.setTfLCOperationEntryParameters(...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 30, ...
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName', 'matrix_sum_4x4_double');
```

Create the conceptual representation of the input argument by creating an `RTW.TfLArgMatrix` object. Use the object to specify properties of the matrix argument. For example, set the dimension range to [4 4; Inf Inf]. The argument must be a two-dimensional matrix of size 4-by-4 or larger.

```
arg = RTW.TfLArgMatrix('u1', 'RTW_IO_INPUT', 'single');
arg.DimRange = [4 4; Inf Inf];
```

Add the conceptual argument to the entry.

```
hEntry.addConceptualArg(arg);
```

After you create the entry, add the entry to the code replacement table.

Version History

Introduced in R2007b

See Also

`addConceptualArg` | `RTW.TfLCOperationEntry`

Topics

"Matrix Multiplication Operation to MathWorks BLAS Code Replacement"

"Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement"

schedule

Namespace: coder.profile

Visualize task scheduling

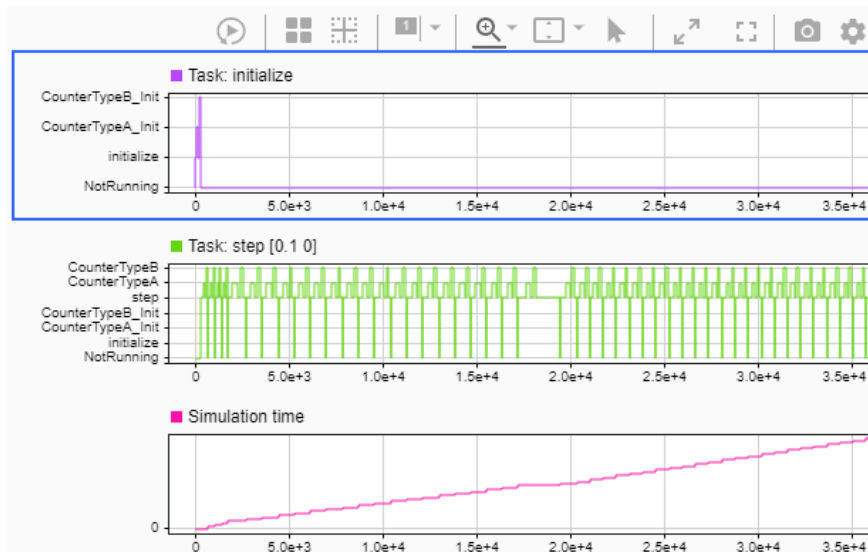
Syntax

```
schedule(executionProfile)
schedule(executionProfile,Name,Value)
```

Description

`schedule(executionProfile)`, using the Simulation Data Inspector, helps you to visualize how code was executed on the target hardware in the last software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulation.

`schedule(executionProfile,Name,Value)` uses name-value arguments to control the display of function execution and simulation time.



Examples

Visualize Task Scheduling in SIL Simulation

This example uses a SIL simulation to show how you can visualize task scheduling. For an example that uses an XCP external mode simulation, see “Visualize Task Scheduling in XCP External Mode Simulation”.

Run a SIL simulation with a model that is configured to generate a workspace variable with execution-time measurements.


```

openExample('ecoder/SILPILVerificationExample', ...
    supportingFile='SILTopModel.slx')

% Disable Simulink Code Coverage and third-party code coverage analysis
set_param('SILTopModel',...
    'CovEnable', 'off');
covSettings = get_param('SILTopModel', 'CodeCoverageSettings');
covSettings.CoverageTool = 'None';
set_param('SILTopModel', 'CodeCoverageSettings', covSettings);

% Configure code execution time profiling
set_param('SILTopModel',...
    'CodeExecutionProfiling', 'on');
set_param('SILTopModel',...
    'CodeProfilingInstrumentation', 'Detailed');
set_param('SILTopModel',...
    'CodeProfilingSaveOptions', 'AllData');
simOut = sim('SILTopModel');

```

The simulation generates the variable `executionProfile` (default) in the object `simOut`.

At the end of the SIL simulation, open the Code Profile Analyzer:

- 1 Click the **SIL/PIL** tab.
- 2 In the **Results** gallery, under **Execution Profiling Results**, click **Code Profile Analyzer**.

In the **Analysis** section, click **Function Execution**. You see execution-time metrics for six code sections, for example, the `step [0.1 0]` task and the `CounterTypeA` and `CounterTypeB` functions.

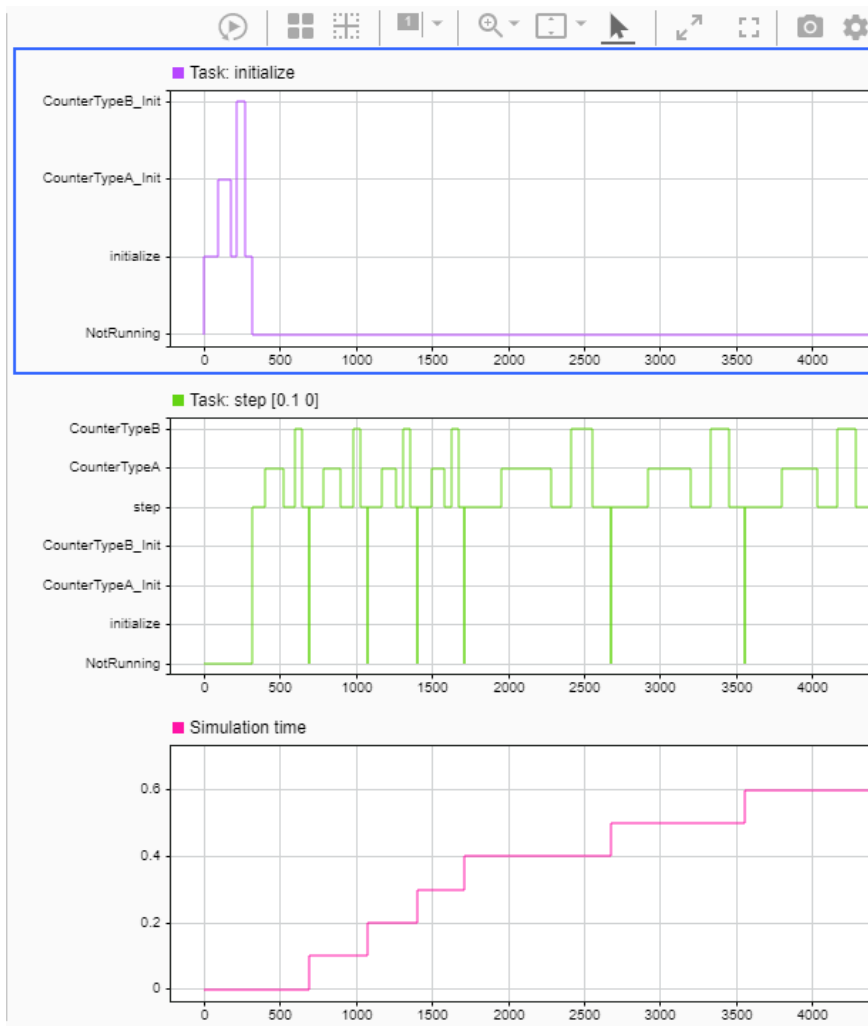
Section	Minimum Execution Time	Average Execution Time	Maximum Execution Time	Average Self Time	Maximum Self Time	Calls	Execution Time Standard Deviation	Execution Time Variance
initialize	62.7778	62.7778	62.7778	30.5556	30.5556	1	0	0
CounterTypeA	18.8889	18.8889	18.8889	18.8889	18.8889	1	0	0
CounterTypeB	13.3333	13.3333	13.3333	13.3333	13.3333	1	0	0
step [0.1 0]	87.7778	141.8537	437.7778	85.5226	378.8889	101	55.1524	1.0950e+04
CounterTypeA	20.5556	35.0825	158.8889	35.0825	158.8889	101	21.0737	1.5988e+03
CounterTypeB	10.0000	21.2486	113.3333	21.2486	113.3333	101	13.4036	646.7646

Section	Self Time / Caller (%)	Self Time / Task (%)	Self Time / Simulation (%)	Function / Simulation (%)
initialize	48.6726	48.6726	0.2123	0.4363
CounterTypeA	30.0885	30.0885	0.1313	0.1313
CounterTypeB	21.2389	21.2389	0.0927	0.0927
step [0.1 0]	60.2893	60.2893	60.0263	99.5637
CounterTypeA	24.7315	24.7315	24.6236	24.6236
CounterTypeB	14.9793	14.9793	14.9139	14.9139

To visualize how the tasks are scheduled and generated code is executed, in the **Results** section, click **Generate Schedule**. Or, in the Command Window, enter:

```
schedule(simOut.executionProfile)
```

The Simulation Data Inspector displays task and simulation time plots.



In each task plot, the Y-axis lists tasks and functions executed. From the plots, you can infer the following information:

- The order in which tasks run.

If the model is multi-rate, you can see how Simulink schedules the different rates (a task for each rate).

- The time that is required to execute a task or a function, computed as the difference between the stop and start times. For example, observe that **CounterTypeB** takes less time to run than **CounterTypeA**. When a task is not running, the Y-axis value of the plot is **NotRunning**.
- The order in which functions run within a task. For example, in the initialize task, **counterTypeA_Init** function runs before **counterTypeB_Init** function. If function calls are nested, you can see the execution order of the functions.

The last plot shows the simulation time when the tasks and functions are executed.

Input Arguments

executionProfile — Variable with profiling data

object

Variable specified by the **Workspace variable** configuration parameter, which contains the code execution profiling data. The SIL or PIL simulation generates the variable.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `schedule(simOut.executionProfile, 'ShowTasksOnly', false, 'StartSimTime', 0.1, 'StopSimTime', 5.7)`

ShowTasksOnly — Turn off function execution display

false (default) | true

Control display of function execution plots:

- `true` -- Display task execution only. Do not display function execution.
- `false` -- Display task and function execution.

MaxNumPoints — Maximum number of display points

integer

Specify maximum number of points to display.

StartSimTime — Simulation time at start of display

float

Specify simulation time at the start of the display.

StopSimTime — Simulation time at end of display

float

Specify simulation time at the end of the display.

Version History

Introduced in R2021a

See Also

`report` | `SIL/PIL Manager` | `Code Profile Analyzer`

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Visualize Task Scheduling in XCP External Mode Simulation”

“SIL/PIL Manager Verification Workflow”

Simulation Data Inspector

schedule

Namespace: `coder.profile.mc`

Visualize task scheduling (MATLAB code generation)

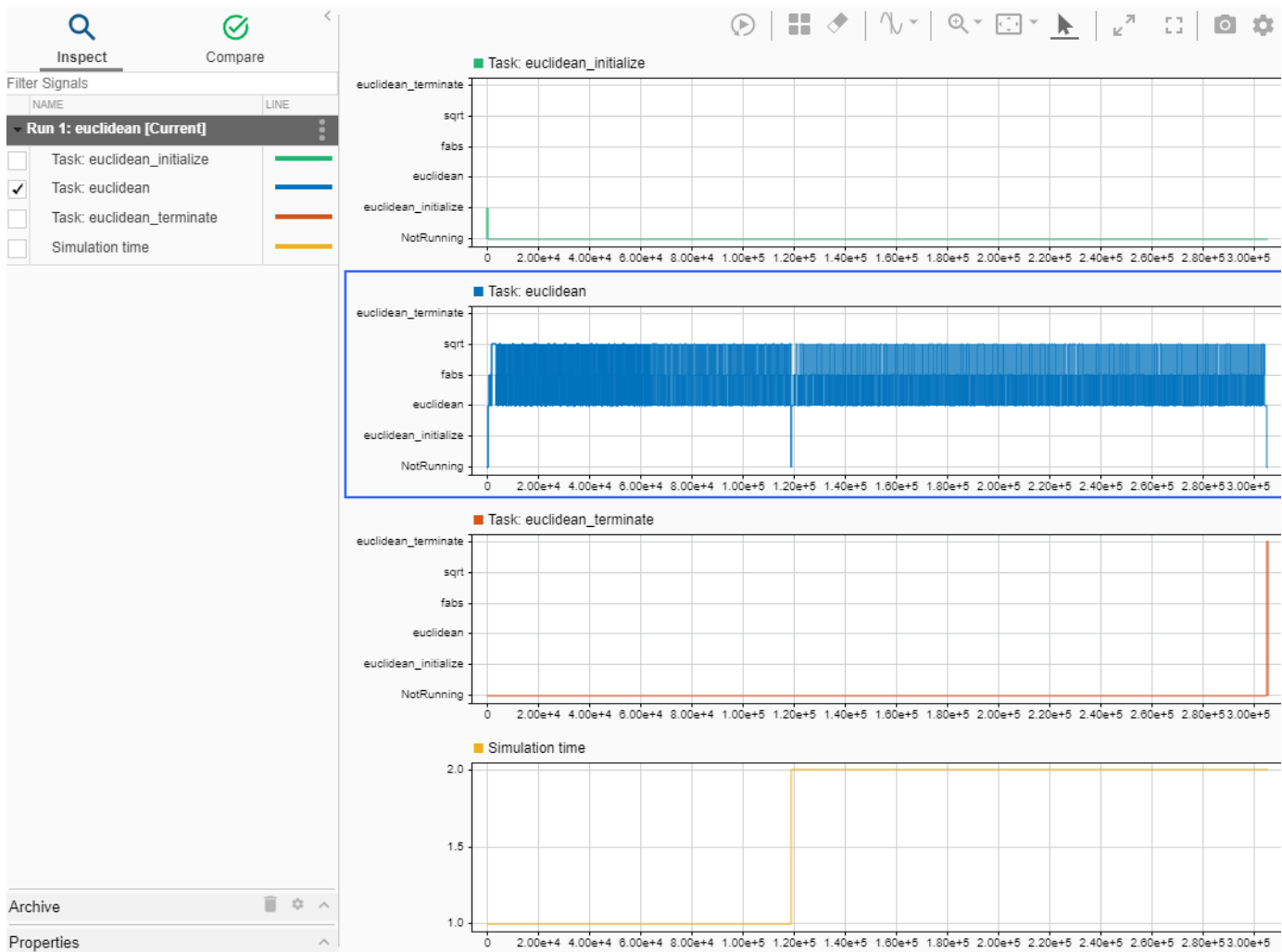
Syntax

```
schedule(executionProfile)  
schedule(executionProfile,Name,Value)
```

Description

`schedule(executionProfile)`, using the Simulation Data Inspector, helps you to visualize how code was executed on the target hardware in the last software-in-the-loop (SIL) or processor-in-the-loop (PIL) run.

`schedule(executionProfile,Name,Value)` uses name-value arguments to control the display of function execution and execution time.



Examples

Visualize Task Scheduling

For a MATLAB function, perform a software-in-the-loop execution that generates execution-time measurements and then use the data to visualize task scheduling.

Open the `EuclideanDistanceExample` example to obtain the files for this tutorial. For more information about this example, see “Generate Standalone C Static Library at the Command Line”.

```
openExample('coder/EuclideanDistanceExample')
```

To make the MATLAB code suitable for code generation, in `euclidean.m`, replace:

```
idx(1)=1;
idx(2)=1;

distance(1)=norm(x-cb(:,1));
distance(2)=norm(x-cb(:,1));
```

with:

```
idx = ones(1,2);
distance = ones(1,2)*norm(x-cb(:,1));
```

Configure SIL execution with profiling.

```
config = coder.config('lib');
config.GenerateReport = true;
config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;
config.CodeProfilingInstrumentation = true;
config.EnableOpenMP = false;
```

In the current folder, create `test_example.m`, a MATLAB script that calls the compiled function twice.

```
[y_min,y_max,idx,distance] = euclidean(x,cb);
[y_min2,y_max2,idx2,distance2] = euclidean(x*2,cb*2);
```

Generate library code for the `euclidean` MATLAB function and the SIL interface and start the SIL execution.

```
load euclidean_data.mat
codegen -report -config config euclidean -args {x, cb} -test test_example
```

Code generation successful: [View report](#)

```
Running test file: 'test' with MEX function 'euclidean_sil.mexw64'.
### Starting SIL execution for 'euclidean'
    To terminate execution: clear euclidean_sil
    Execution profiling data is available for viewing. Go to Simulation Data Inspector.
    Execution profiling report available after termination.
```

...

Terminate the SIL execution process by clicking the link `clear euclidean_sil`.

```
### Application stopped
### Stopping SIL execution for 'euclidean'
    Execution profiling report: report(getCoderExecutionProfile('euclidean'))
```

Create a workspace variable that holds the execution time data.

```
executionProfile=getCoderExecutionProfile('euclidean');
```

To open the code execution report, run:

```
report(executionProfile)
```

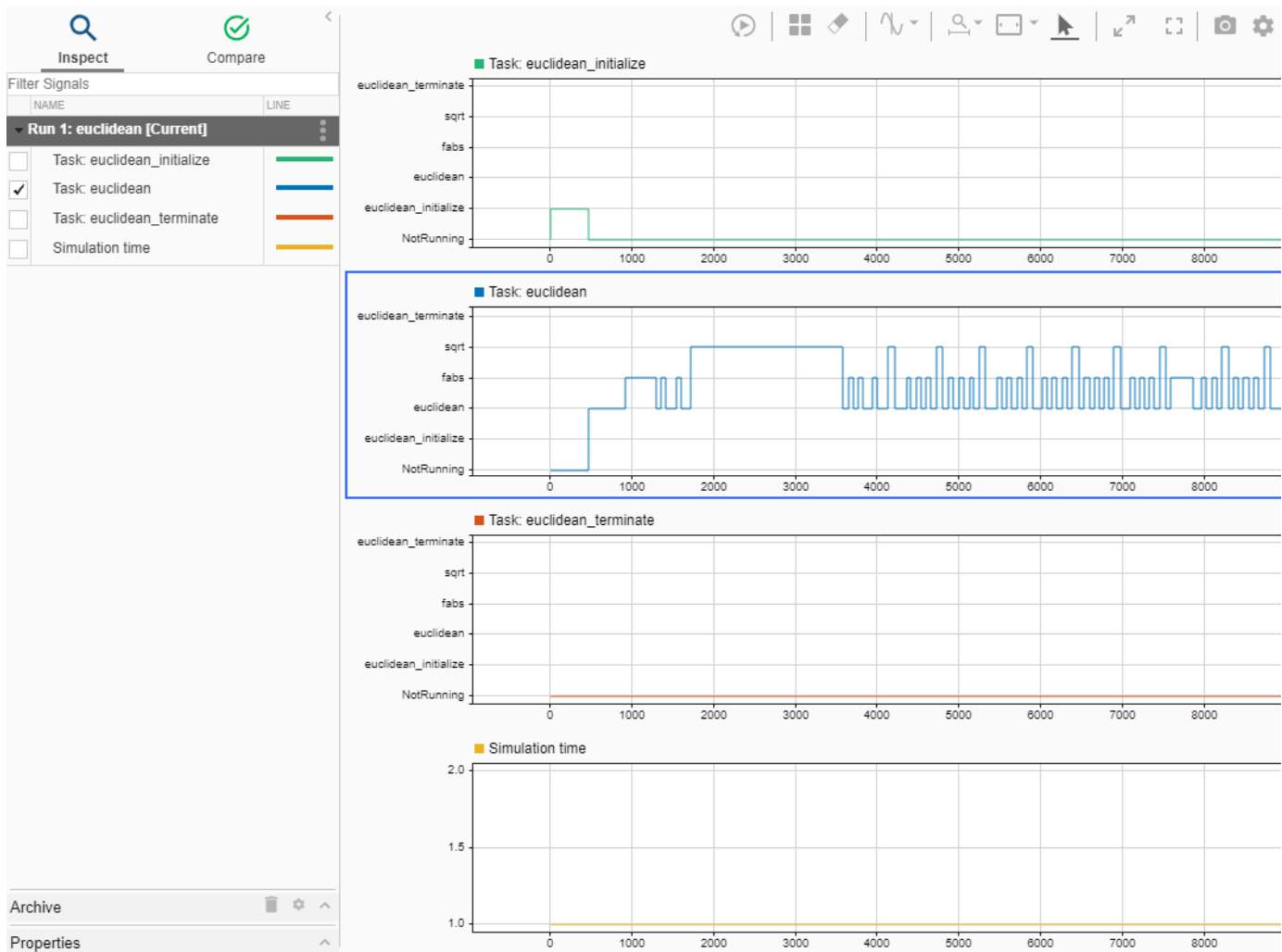
2. Profiled Sections of Code

Section	Maximum Execution Time in ns	Average Execution Time in ns	Maximum Self Time in ns	Average Self Time in ns	Calls	
euclidean_initialize	129	129	129	129	1	 
[+] euclidean	51657	42318	26931	21927	2	   
euclidean_terminate	98	98	98	98	1	 

To visualize how the tasks are scheduled and generated code is executed, run:

```
schedule(executionProfile)
```

The Simulation Data Inspector displays these plots.



In each task plot, the Y-axis lists tasks and functions called by each task. From the plots, you can infer the following information:

- The order in which tasks run. For example, `initialize` runs before `euclidean`.
- The time that is required to execute a task or a function, computed as the difference between the stop and start times. For example, observe that the initial calls to `fabs` and `sqrt` take more time to run than subsequent calls to the same functions. When a task is not running, the Y-axis value of the plot is `NotRunning`.
- The order in which functions run within a task. For example, in the `euclidean` task, there are three calls to `fabs` before each call to `sqrt`.

If function calls are nested, you can see the call stack for the functions.

The simulation time plot displays the call number for the `euclidean` function. In this example, there are two calls to `euclidean`.

Input Arguments

executionProfile — Variable with profiling data

object

Workspace variable that contains the code execution profiling data generated by the SIL or PIL execution.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `schedule(executionProfile, 'ShowTasksOnly', false, 'StartSimTime', 2, 'StopSimTime', 4)`

ShowTasksOnly — Turn off function execution display

false (default) | true

Control display of function execution plots:

- `true` -- Display task execution only. Do not display function execution.
- `false` -- Display task and function execution.

MaxNumPoints — Maximum number of display points

integer

Specify maximum number of points to display.

StartSimTime — Execution time at start of display

float

Specify execution time at the start of the display.

StopSimTime — Execution time at end of display

float

Specify execution time at the end of the display.

Version History

Introduced in R2021b

See Also

`getCoderExecutionProfile` | `report`

Topics

"Generate Execution Time Profile"

"View Execution Times"

"Generate Standalone C Static Library at the Command Line"

Simulation Data Inspector

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

If you have a model `myModel.slx` that can run a SIL, PIL, or XCP-based external mode simulation, you can configure the model to generate a workspace variable that contains execution time measurements.

```
set_param('myModel', 'CodeExecutionProfiling', 'on');
set_param('myModel', 'CodeProfilingInstrumentation', 'detailed');
set_param('myModel', 'CodeProfilingSaveOptions', 'AllData');
```

When you run the command `sim('myModel')`, the simulation generates the variable `executionProfile` (default). At the end of the simulation, get the profile for a code section. For example:

```
seventhSectionProfile = executionProfile.Sections(7);
```

Then, get the vector that represents simulation time for the code section.

```
simulationTimeVector = seventhSectionProfile.Time;
```

In this example, run a SIL model simulation where the model generates a workspace variable that contains execution time measurements.

```
openExample('ecoder/SILPILVerificationExample', ...
    supportingFile='SILTopModel.slx')

% Disable Simulink Code Coverage and third-party code coverage analysis
set_param('SILTopModel', ...
    'CovEnable', 'off');
covSettings = get_param('SILTopModel', 'CodeCoverageSettings');
covSettings.CoverageTool = 'None';
set_param('SILTopModel', 'CodeCoverageSettings', covSettings);

% Configure code execution time profiling
set_param('SILTopModel', ...
    'CodeExecutionProfiling', 'on');
set_param('SILTopModel', ...
    'CodeProfilingInstrumentation', 'detailed');
set_param('SILTopModel', ...
    'CodeProfilingSaveOptions', 'AllData');
simOut = sim('SILTopModel');
```

The simulation generates the variable `executionProfile` (default) in the object `simOut`.

At the end of the simulation, get the profile for the sixth code section.

```
sixthSectionProfile = simOut.executionProfile.Sections(6);
```

Get the vector that represents simulation time for the code section.

```
simulationTimeVector = sixthSectionProfile.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.

Version History

Introduced in R2013a

See Also

`Sections` | `ExecutionTimeInTicks` | `ExecutionTimeInSeconds`

Topics

“Create Execution-Time Profile for Generated Code”

“Analyze Code Execution Data”

Time

Namespace: `coder.profile.mc`

Time over which code section execution time measurements are made (MATLAB code generation)

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

Open this example to obtain the files for this tutorial:

- `kalman01.m` — MATLAB function for the Kalman estimator
- `test01_ui.m` — MATLAB file to test `kalman01.m`
- `plot_trajectory.m` — File that plots actual target trajectory and Kalman estimator output
- `position.mat` — Input data

```
openExample('ecoder/KalmanFilterExample')
```

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.

Version History

Introduced in R2013a

See Also

`getCoderExecutionProfile` | `Sections` | `ExecutionTimeInTicks` | `ExecutionTimeInSeconds`

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

TimerTicksPerSecond

Namespace: `coder.profile`

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 . You can calculate the execution time in seconds using the formula *ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond*.

`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.

Examples

Get Timer Ticks per Second Value

To get the number of timer ticks per second, get the `TimerTicksPerSecond` property value from the `myExecutionProfile` workspace variable.

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond;
```

Set Timer Ticks per Second Value

To set the number of timer ticks per second, set the `TimerTicksPerSecond` property value in the `myExecutionProfile` workspace variable.

```
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal;
```

Input Arguments

myExecutionProfile — Workspace variable
workspace variable

`myExecutionProfile` is a workspace variable generated by a simulation.

Example: `myExecutionProfile`

timerTicksPerSecVal — Number of timer ticks
number of timer ticks

Number of timer ticks per second

Example: `timerTicksPerSecVal`

Output Arguments

timerTicksPerSecVal — Number of timer ticks

number of timer ticks

Number of timer ticks per second

Version History

Introduced in R2012b

See Also

`Sections` | `display` | `report` | `Name` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` | `TotalExecutionTimeInTicks` | `SelfTimeInTicks` | `MaximumSelfTimeInTicks` | `TotalSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `TurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks`

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

TimerTicksPerSecond

Namespace: `coder.profile.mc`

Get and set number of timer ticks per second (MATLAB code generation)

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 .

You can calculate the execution time in seconds using the formula
ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond.

`myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal` sets the number of timer ticks per second. Use this method if the target connectivity configuration does not specify this value.

Examples

Get Timer Ticks per Second Value

To get the number of timer ticks per second, get the `TimerTicksPerSecond` property value from the `myExecutionProfile` workspace variable.

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond;
```

Set Timer Ticks per Second Value

To set the number of timer ticks per second, set the `TimerTicksPerSecond` property value in the `myExecutionProfile` workspace variable.

```
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal;
```

Input Arguments

myExecutionProfile — Workspace variable that contains the execution-time profile
workspace variable

The `myExecutionProfile` is a workspace variable that contains the execution-time profile of the code generated from your MATLAB function.

timerTicksPerSecVal — Number of timer ticks

number of timer ticks

Number of timer ticks per second

Example: timerTicksPerSecVal

Output Arguments**timerTicksPerSecVal — Number of timer ticks**

number of timer ticks

Number of timer ticks per second

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | report | Name | Number | NumCalls |
MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

“Create PIL Target Connectivity Configuration for MATLAB”

coder.MATLABCodeTemplate.getTokenValue

Class: coder.MATLABCodeTemplate

Namespace: 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.setTokenValue` |
`coder.MATLABCodeTemplate.getcurrenttokens` |
`coder.MATLABCodeTemplate.emitSection`

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

If you have a model `myModel.slx` that can run a SIL, PIL, or XCP-based external mode simulation, you can configure the model to generate a workspace variable that contains execution time measurements.

```
set_param('myModel', 'CodeExecutionProfiling', 'on');  
set_param('myModel', 'CodeProfilingInstrumentation', 'detailed');  
set_param('myModel', 'CodeProfilingSaveOptions', 'AllData');
```

When you run the command `sim('myModel')`, the simulation generates the variable `executionProfile` (default). At the end of the simulation, get the profile for a code section. For example:

```
seventhSectionProfile = executionProfile.Sections(7);
```

Then, get the vector of execution times for the code section.

```
time_vector = seventhSectionProfile.ExecutionTimeInSeconds;
```

In this example, run a SIL model simulation where the model generates a workspace variable that contains execution time measurements.

```
openExample('ecoder/SILPILVerificationExample', ...  
    supportingFile='SILTopModel.slx')  
  
% Disable Simulink Code Coverage and third-party code coverage analysis  
set_param('SILTopModel', ...  
    'CovEnable', 'off');  
covSettings = get_param('SILTopModel', 'CodeCoverageSettings');  
covSettings.CoverageTool = 'None';  
set_param('SILTopModel', 'CodeCoverageSettings', covSettings);
```

```
% Configure code execution time profiling
set_param('SILTopModel',...
    'CodeExecutionProfiling', 'on');
set_param('SILTopModel',...
    'CodeProfilingInstrumentation', 'detailed');
set_param('SILTopModel',...
    'CodeProfilingSaveOptions', 'AllData');
simOut = sim('SILTopModel');
```

The simulation generates the variable `executionProfile` (default) in the object `simOut`.

At the end of the simulation, get the profile for the sixth code section.

```
sixthSectionProfile = simOut.executionProfile.Sections(6);
```

Get the vector of execution times for the code section.

```
time_vector = sixthSectionProfile.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.

Version History

Introduced in R2013a

See Also

`Sections` | `ExecutionTimeInTicks`

Topics

“Create Execution-Time Profile for Generated Code”
“Analyze Code Execution Data”

ExecutionTimeInSeconds

Namespace: `coder.profile.mc`

Get execution time in seconds for profiled section of code (MATLAB code generation)

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

Open this example to obtain the files for this tutorial:

- `kalman01.m` — MATLAB function for the Kalman estimator
- `test01_ui.m` — MATLAB file to test `kalman01.m`
- `plot_trajectory.m` — File that plots actual target trajectory and Kalman estimator output
- `position.mat` — Input data

```
openExample('ecoder/KalmanFilterExample')
```

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.

Version History

Introduced in R2013a

See Also

`getCoderExecutionProfile` | `Sections` | `ExecutionTimeInTicks`

Topics

"Generate Execution Time Profile"

"Analyze Execution Time Data"

ExecutionTimeInTicks

Namespace: `coder.profile`

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.

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.

You can calculate the execution time in seconds using the formula
ExecutionTimeInSecs = ExecutionTimeInTicks/TimerTicksPerSecond

Examples

Get Execution Time in Ticks

If you have a model `myModel.slx` that can run a SIL, PIL, or XCP-based external mode simulation, you can configure the model to generate a workspace variable that contains execution time measurements.

```
set_param('myModel', 'CodeExecutionProfiling', 'on');  
set_param('myModel', 'CodeProfilingInstrumentation', 'detailed');  
set_param('myModel', 'CodeProfilingSaveOptions', 'AllData');
```

When you run the command `sim('myModel')`, the simulation generates the variable `executionProfile` (default). At the end of the simulation, get the profile for a code section. For example:

```
seventhSectionProfile = executionProfile.Sections(7);
```

Then, get the vector of execution times (measured in timer ticks) for the code section.

```
time_vector = seventhSectionProfile.ExecutionTimeInTicks;
```

In this example, run a SIL model simulation where the model generates a workspace variable that contains execution time measurements.


```

openExample('ecoder/SILPILVerificationExample', ...
    supportingFile='SILTopModel.slx')

% Disable Simulink Code Coverage and third-party code coverage analysis
set_param('SILTopModel',...
    'CovEnable', 'off');
covSettings = get_param('SILTopModel', 'CodeCoverageSettings');
covSettings.CoverageTool = 'None';
set_param('SILTopModel', 'CodeCoverageSettings', covSettings);

% Configure code execution time profiling
set_param('SILTopModel',...
    'CodeExecutionProfiling', 'on');
set_param('SILTopModel',...
    'CodeProfilingInstrumentation', 'detailed');
set_param('SILTopModel',...
    'CodeProfilingSaveOptions', 'AllData');
simOut = sim('SILTopModel');

```

The simulation generates the variable `executionProfile` (default) in the object `simOut`.

At the end of the simulation, get the profile for the sixth code section.

```
sixthSectionProfile = simOut.executionProfile.Sections(6);
```

Get the vector of execution times (measured in timer ticks) for the code section.

```
time_vector = sixthSectionProfile.ExecutionTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

ExecutionTimes — Vector of execution times

vector of execution times

The `SelfExecutionTimes` is a vector of execution times, in timer ticks, for profiled section of code.

Version History

Introduced in R2012b

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `MaximumExecutionTimeInTicks` | `TotalExecutionTimeInTicks` | `SelfTimeInTicks` | `MaximumSelfTimeInTicks` | `TotalSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `TurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks`

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”
“Analyze Code Execution Data”

ExecutionTimeInTicks

Namespace: `coder.profile.mc`

Get execution times in timer ticks for profiled section of code (MATLAB code generation)

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.

Examples

Get Execution Time in Ticks

To get a vector of execution times, measured in timer ticks, for the profiled section of code, use the `ExecutionTimeInTicks` property of the `NthSectionProfile` object.

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

ExecutionTimes — Vector of execution times

vector of execution times

The `SelfExecutionTimes` is a vector of execution times, in timer ticks, for profiled section of code.

Version History

Introduced in R2012b

See Also

ExecutionTimeInSeconds | getCoderExecutionProfile | Sections | TimerTicksPerSecond
| report | Name | Number | NumCalls | MaximumExecutionTimeCallNum |
MaximumSelfTimeCallNum | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks
| SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum |
TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”
“Analyze Execution Time Data”

MaximumExecutionTimeCallNum

Namespace: `coder.profile`

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.

Examples

Get Maximum Execution Time Call Number

To get the call number at which the maximum number of timer ticks was recorded, use the `MaximumExecutionTimeCallNum` property of the `NthSectionProfile` object.

```
MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxTicksCallNum — Call number at which the maximum number of timer ticks occurred

integer

The `MaxTicksCallNum` is the call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | ExecutionTimeInTicks |
Number | NumCalls | MaximumSelfTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

MaximumExecutionTimeCallNum

Namespace: `coder.profile.mc`

Get the call number at which maximum number of timer ticks occurred (MATLAB code generation)

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.

Examples

Get Maximum Execution Time Call Number

To get the call number at which the maximum number of timer ticks was recorded, use the `MaximumExecutionTimeCallNum` property of the `NthSectionProfile` object.

```
MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxTicksCallNum — Call number at which the maximum number of timer ticks occurred

integer

The `MaxTicksCallNum` is the call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

Version History

Introduced in R2012b

See Also

`getCoderExecutionProfile` | `Sections` | `TimerTicksPerSecond` | `report` | `Name` | `Number` | `NumCalls` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` | `TotalExecutionTimeInTicks` | `SelfTimeInTicks` | `MaximumSelfTimeInTicks` | `TotalSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `TurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks`

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

MaximumExecutionTimeInTicks

Namespace: `coder.profile`

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.

Examples

Get Maximum Timer Ticks Recorded

Get the maximum number of timer ticks recorded in a single invocation of the profiled code section during a simulation.

```
MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks;
```

Input Arguments

NthSectionProfile — ExecutionTimeSection object

`ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile.MaximumExecutionTimeInTicks`

Output Arguments

MaxTicks — Maximum number of timer ticks

integer

Maximum number of timer ticks for single invocation of profiled code section

Version History

Introduced in R2012b

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `ExecutionTimeInTicks` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` |

ExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

MaximumExecutionTimeInTicks

Namespace: `coder.profile.mc`

Get maximum number of timer ticks for single invocation of profiled code section (MATLAB code generation)

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.

Examples

Get Maximum Timer Ticks Recorded

Get the maximum number of timer ticks recorded in a single invocation of the profiled code section.

```
MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks;
```

Input Arguments

NthSectionProfile — ExecutionTimeSection object

ExecutionTimeSection object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile.MaximumExecutionTimeInTicks`

Output Arguments

MaxTicks — Maximum timer ticks single invocation

integer

The `MaxTicks` is the maximum number of timer ticks for single invocation of profiled code section.

Version History

Introduced in R2012b

See Also

`getCoderExecutionProfile` | `Sections` | `TimerTicksPerSecond` | `report` | `Name` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` |

ExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

TotalExecutionTimeInTicks

Namespace: `coder.profile`

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.

Examples

Get Total Execution Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the `TotalTicks` property of the `NthSectionProfile` object.

```
TotalTicks = NthSectionProfile.TotalTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalTicks — Total number of timer ticks

total number of timer ticks

The `TotalTicks` is the total number of timer ticks for profiled code section.

Version History

Introduced in R2012b

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` |

MaximumExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks |
TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

TotalExecutionTimeInTicks

Namespace: `coder.profile.mc`

Get total number of timer ticks recorded for profiled code section (MATLAB code generation)

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.

Examples

Get Total Execution Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the `TotalTicks` property of the `NthSectionProfile` object.

```
TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalTicks — Total number of timer ticks

total number of timer ticks

The `TotalTicks` is the total number of timer ticks for profiled code section.

Version History

Introduced in R2012b

See Also

`getCoderExecutionProfile` | `Sections` | `TimerTicksPerSecond` | `report` | `Name` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum` |

ExecutionTimeInTicks | MaximumExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

SelfTimeInTicks

Namespace: `coder.profile`

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.

Examples

Get Execution Self Time in Ticks

To get the number of timer ticks recorded for the profiled code section, use the `SelfTimeInTicks` property of the `NthSectionProfile` object.

```
SelfTicks = NthSectionProfile.SelfTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

SelfTicks — Number of timer ticks for profiled code section

integer

The `SelfTicks` is the number of timer ticks for profiled code section, excluding periods in child functions.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | ExecutionTimeInTicks |
Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum |
ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

SelfTimeInTicks

Namespace: `coder.profile.mc`

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions (MATLAB code generation)

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.

Examples

Get Execution Self Time in Ticks

To get the number of timer ticks recorded for the profiled code section, use the `SelfTimeInTicks` property of the `NthSectionProfile` object.

```
SelfTicks = NthSectionProfile.SelfTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

SelfTicks — Number of timer ticks for profiled code section

integer

The `SelfTicks` is the number of timer ticks for profiled code section, excluding periods in child functions

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | TimerTicksPerSecond | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

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

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | ExecutionTimeInTicks | Number | NumCalls | MaximumExecutionTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

"Create Execution-Time Profile for Generated Code"

"Create Execution-Time Profile for Generated Code"

"View and Compare Code Execution Times"

MaximumSelfTimeCallNum

Namespace: `coder.profile.mc`

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions (MATLAB code generation)

Syntax

`MaxSelfTicksCallNum = NthSectionProfile.MaximumSelfTimeCallNum`

Description

`MaxSelfTicksCallNum = NthSectionProfile.MaximumSelfTimeCallNum` returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

Examples

Get Maximum Execution Time Call Number

To get the call number at which the maximum number of timer ticks was recorded, use the `MaximumExecutionTimeCallNum` property of the `NthSectionProfile` object.

```
MaxSelfTicksCallNum = NthSectionProfile.MaxSelfTimeCallNum;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxSelfTicksCallNum — Call number of maximum number of self-time ticks

integer

The `MaxSelfTicksCallNum` is the call number at which the maximum number of self-time ticks occurred for profiled code section.

Version History

Introduced in R2012b

See Also

[getCoderExecutionProfile](#) | [TotalTurnaroundTimeInTicks](#)

Topics

[“Generate Execution Time Profile”](#)

[“Analyze Execution Time Data”](#)

MaximumSelfTimeInTicks

Namespace: `coder.profile`

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.

Examples

Get Maximum Execution Self Time in Ticks

To get the maximum number of timer ticks recorded for the profiled code section, use the `MaximumSelfTimeInTicks` property of the `NthSectionProfile` object.

```
MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxSelfTicks — Maximum number of timer ticks

integer

The `MaxSelfTicks` is the maximum number of timer ticks for profiled code section, excluding periods in child functions.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | ExecutionTimeInTicks |
Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum |
ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |
SelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

MaximumSelfTimeInTicks

Namespace: `coder.profile.mc`

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions (MATLAB code generation)

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.

Examples

Get Maximum Execution Self Time in Ticks

To get the maximum number of timer ticks recorded for the profiled code section, use the `MaximumSelfTimeInTicks` property of the `NthSectionProfile` object.

```
MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxSelfTicks — Maximum number of timer ticks

integer

The `MaxSelfTicks` is the maximum number of timer ticks for profiled code section, excluding periods in child functions.

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | TimerTicksPerSecond | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

TotalSelfTimeInTicks

Namespace: `coder.profile`

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.

Examples

Get Execution Self Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the `TotalSelfTimeInTicks` property of the `NthSectionProfile` object.

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalSelfTicks — Value of execution time

value of execution times

The `TotalSelfTicks` is the total number of timer ticks for profiled code section, excluding periods in child functions.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | ExecutionTimeInTicks |
Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum |
ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |
SelfTimeInTicks | MaximumSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

TotalSelfTimeInTicks

Namespace: `coder.profile.mc`

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions (MATLAB code generation)

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.

Examples

Get Execution Self Time in Ticks

To get a value for execution time, measured in timer ticks, for the profiled section of code, use the `TotalSelfTimeInTicks` property of the `NthSectionProfile` object.

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalSelfTicks — Value of execution time

value of execution times

The `TotalSelfTicks` is the total number of timer ticks for profiled code section, excluding periods in child functions.

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | TimerTicksPerSecond | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

MaximumTurnaroundTimeInTicks

Namespace: `coder.profile`

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

`MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks`

Description

`MaxTurnaroundTicks = 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.

Examples

Get Maximum Turnaround Time in Ticks

To get the call number in which the maximum number of timer ticks was recorded, use the `MaximumTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxTurnaroundTicks — Maximum number of timer ticks
integer

The `MaxTurnaroundTicks` is the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number | NumCalls |
MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum |
MaximumTurnaroundTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

MaximumTurnaroundTimeInTicks

Namespace: `coder.profile.mc`

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section (MATLAB code generation)

Syntax

`MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks`

Description

`MaxTurnaroundTicks = 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 an execution. Unless the code is pre-empted, this is the same as the maximum execution time.

Examples

Get Maximum Turnaround Time in Ticks

To get the call number in which the maximum number of timer ticks was recorded, use the `MaximumTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
MaxTurnaroundTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxTurnaroundTicks — Maximum number of timer ticks
integer

The `MaxTurnaroundTicks` is the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | TimerTicksPerSecond | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

MaximumTurnaroundTimeCallNum

Namespace: `coder.profile`

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.

Examples

Get Maximum Turnaround Time Call Number

To get the call number in which the maximum number of timer ticks was recorded, use the `MaximumTurnaroundTimeCallNum` property of the `NthSectionProfile` object.

```
MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object
`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Example: `NthSectionProfile`

Output Arguments

MaxTurnaroundTicksCallNum — Call number of the maximum number of timer ticks
integer

The `MaxTurnaroundTicksCallNum` is the call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number | NumCalls |
MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | TurnaroundTimeInTicks |
MaximumTurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

MaximumTurnaroundTimeCallNum

Namespace: `coder.profile.mc`

Get call number for the code section invocation with the maximum number of timer ticks between the start and the finish (MATLAB code generation)

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.

Examples

Get Maximum Turnaround Time Call Number

To get the call number in which the maximum number of timer ticks was recorded, use the `MaximumTurnaroundTimeCallNum` property of the `NthSectionProfile` object.

```
MaxTurnaroundTicksCallNum = NthSectionProfile.MaximumTurnaroundTimeCallNum;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

MaxTurnaroundTicksCallNum — Call number of the maximum number of timer ticks

integer

The `MaxTurnaroundTicksCallNum` is the call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section.

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | TimerTicksPerSecond | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks | TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

TotalTurnaroundTimeInTicks

Namespace: `coder.profile`

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.

Examples

Get Total Turnaround Time in Ticks

To get the total number of timer ticks recorded for turnaround time, use the `TotalTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalTurnaroundTicks — Total number of timer ticks between start and finish

integer

The `TotalTurnaroundTicks` is the total number of timer ticks between start and finish of the profiled code section over the entire simulation.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number | NumCalls |
MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum |
MaximumTurnaroundTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | TurnaroundTimeInTicks |
MaximumTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

TotalTurnaroundTimeInTicks

Namespace: `coder.profile.mc`

Get total number of timer ticks between start and finish of the profiled code section over the entire execution. (MATLAB code generation)

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.

Examples

Get Total Turnaround Time in Ticks

To get the total number of timer ticks recorded for turnaround time, use the `TotalTurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TotalTurnaroundTicks — Total number of timer ticks between start and finish

integer

The `TotalTurnaroundTicks` is the total number of timer ticks between start and finish of the profiled code section over the entire simulation.

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | TimerTicksPerSecond | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

TurnaroundTimeInTicks

Namespace: `coder.profile`

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.

Examples

Get Turnaround Time in Ticks

To get the number of timer ticks recorded for turnaround time, use the `TurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TurnaroundTicks — Number of timer ticks between start and finish

integer

The `TurnaroundTicks` is the number of timer ticks between start and finish of the profiled code section.

Version History

Introduced in R2012b

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number | NumCalls |
MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum |
MaximumTurnaroundTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks |
MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Topics

“Create Execution-Time Profile for Generated Code”
“View and Compare Code Execution Times”
“Analyze Code Execution Data”

TurnaroundTimeInTicks

Namespace: `coder.profile.mc`

Get number of timer ticks between start and finish of the profiled code section (MATLAB code generation)

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.

Examples

Get Turnaround Time in Ticks

To get the number of timer ticks recorded for turnaround time, use the `TurnaroundTimeInTicks` property of the `NthSectionProfile` object.

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection` object

`coder.profile.ExecutionTimeSection` object

The `NthSectionProfile` is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Example: `NthSectionProfile`

Output Arguments

TurnaroundTicks — Number of timer ticks between start and finish

integer

The `TurnaroundTicks` is the number of timer ticks between start and finish of the profiled code section

Version History

Introduced in R2012b

See Also

getCoderExecutionProfile | Sections | TimerTicksPerSecond | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum | TotalTurnaroundTimeInTicks

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

modifyInheritedParam

Namespace: `rtw.codegenObjectives`

Modify inherited parameter values in code generation objective

Syntax

`modifyInheritedParam(objective, param, value)`

Description

`modifyInheritedParam(objective, param, value)` changes the value of the specified inherited parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**. Use `modifyInheritedParam` to change the value of a parameter in an objective that you created from an existing objective.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the Traceability objective.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace', 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConvertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');

% Exclude the inherited check for the software environment
```



```
excludeCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');  
%Register the objective  
register(obj);  
end
```

Input Arguments

objective — Code generation objective

rtw.codegenObjectives.Objective object

Code generation objective, specified as a rtw.codegenObjectives.Objective object.

param — Name of inherited parameter

character vector | string scalar

Name of inherited parameter to modify, specified as a character vector or string scalar.

value — Parameter value

character vector | string scalar

Parameter value to verify in the Code Generation Advisor, specified as a character vector or string scalar.

Version History

Introduced in R2009a

See Also

get_param

Topics

“Create Custom Code Generation Objectives”

plot

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

Namespace: `rtw.codegenObjectives`

Register objective in Code Generation Advisor

Syntax

`register(objective)`

Description

`register(objective)` registers the specified objective to the end of the list of available objectives that you can use with the Code Generation Advisor.

Examples

Create a Custom Code Generation Objective

Create a custom objective named `Reduced RAM Example` that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

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

```
%Register the objective  
register(obj);  
  
end
```

Input Arguments

objective — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

Version History

Introduced in R2009a

See Also

Topics

“Create Custom Code Generation Objectives”

“Register Customizations with 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`, ..., `un`
 - 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.

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.

Version History

Introduced in R2007b

See Also

`registerCPromotableMacroEntry`

Topics

“Define Code Replacement Library Optimizations”

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*, ..., *un*
 - 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 Library”.

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.

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.

Version History

Introduced in R2010a

See Also

`enableCPP` | `setNameSpace`

Topics

"Define Code Replacement Library Optimizations"

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`, ..., `un`
 - 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</math>', '', '');
```

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.

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.

Version History

Introduced in R2007b

See Also

`registerCFunctionEntry`

Topics

“Define Code Replacement Library Optimizations”

removeInheritedCheck

Namespace: rtw.codegenObjectives

Remove inherited check from objective

Syntax

```
removeInheritedCheck(objective, checkID)
```

Description

`removeInheritedCheck(objective, checkID)` removes the specified check that the objective inherited from the objective that you used to create it. Use this method when you create a new objective from an existing objective. If you select multiple objectives in the Code Generation advisor, if another selected objective includes this check, the advisor includes the check.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the Traceability objective.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace',, 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConvertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');

% Exclude the inherited check for the software environment
```

```
excludeCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');  
  
%Register the objective  
register(obj);  
  
end
```

Input Arguments

objective — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

checkID — Identifier of check

character vector | string scalar

Identifier of check that you want to remove, specified as a character vector or string scalar.

Example: `'mathworks.codegen.CodeInstrumentation'`

Version History

Introduced in R2009a

See Also

`Simulink.ModelAdvisor`

Topics

“Create Custom Code Generation Objectives”

`Simulink.ModelAdvisor`

removeInheritedParam

Namespace: rtw.codegenObjectives

Remove inherited parameter from objective

Syntax

```
removeInheritedParam(objective, param)
```

Description

`removeInheritedParam(objective, param)` removes the specified inherited parameter from the objective. Use this method when you create a new objective from an existing objective. When you select multiple objectives in the Code Generation advisor, if another selected objective includes the parameter, the advisor reviews the parameter value using **Check model configuration settings against code generation objectives**.

Examples

Create an Objective Based On an Existing Objective

Create a custom objective based on the Traceability objective.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and modify the list of checks, parameters, and values to verify. Then register the objective in the Code Generation Advisor.

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_trace', 'Traceability');
setObjectiveName(obj, 'Custom Traceability Example');

% Remove inherited parameters from the objective
removeInheritedParam(obj, 'MATLABFcnDesc');
removeInheritedParam(obj, 'MATLABSourceComments');

% Remove the inherited code instrumentation check
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');

% Modify the inherited parameter 'ConvertIfToSwitch' and set the value to 'on'
modifyInheritedParam(obj, 'ConvertIfToSwitch', 'on');
```

```
% Exclude the inherited check for the software environment
excludeCheck(obj, 'mathworks.codegen.SWEnvironmentSpec');

%Register the objective
register(obj);

end
```

Input Arguments

objective — Code generation objective

rtw.codegenObjectives.Objective object

Code generation objective, specified as a rtw.codegenObjectives.Objective object.

param — Name of inherited parameter

character vector | string scalar

Name of inherited parameter to remove, specified as a character vector or string scalar.

Version History

Introduced in R2009a

See Also

get_param

Topics

“Create Custom Code Generation Objectives”

report

Namespace: `coder.profile`

Open code execution profiling report and specify display of time measurements

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, name-value pair)
```

Description

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, name-value pair)` opens the report with display options specified by the name-value character vector pairs.

Examples

Create Report by Using Options

To create a report that displays time in microseconds (10^{-6} seconds) with a precision of three decimal places, select options by using name-value pairs.

```
report(myExecutionProfile, ...
    'Units','Seconds', ...
    'ScaleFactor','1e-06', ...
    'NumericFormat','%0.3f')
```

Create Report Comparing Against Baseline

To create a report that compares execution-time performance against a baseline, use the `baseline` option.

```
report(myExecutionProfile, ...
    'baseline', executionProfileBaseline)
```

Input Arguments

myExecutionProfile — Variable specifies annotation

workspace variable

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

Example: `myExecutionProfile`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `'Units', 'Seconds'`

Units — Time measurement unit

`'Seconds'` (default for SIL simulation on Windows) | `'Ticks'` (default for SIL simulation on non-Windows and PIL simulation unless overridden to `'Seconds'` by number of ticks per second in the target connectivity configuration)

Time measurements displayed in seconds or timer ticks.

Example: `'Units', 'Seconds'`

ScaleFactor — Scale factor for displayed measurements

`'1e-9'` (default) | character vector representation of a number that is a power of 10

To display measurements in microseconds, use a scale factor name-value pair `'ScaleFactor', '1e-6'`. The 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'`.

Example: `'ScaleFactor', '1e-9'`

NumericFormat — Numeric format for displayed measurements

`'%0.0f'` (default) | decimal convention utilized by the ANSI C fallback for ANSI C function `sprintf`

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'`.

Example: `'NumericFormat', '%0.0f'`

baseline — Baseline for performance comparison

`executionProfileBaseline`

The `executionProfileBaseline` is the workspace variable that contains baseline execution-time metrics.

Example: `'baseline', executionProfileBaseline`

Version History

Introduced in R2011b

See Also

`annotate` | `display`

Topics

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

"Code Execution Profiling Report"
"Compare Execution Times Against Baseline"

report

Namespace: `coder.profile.mc`

Open code execution profiling report and specify display of time measurements (MATLAB code generation)

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, name-value pair)
```

Description

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, name-value pair)` opens the report with display options specified by the name-value character vector pairs.

Examples

Create Report by Using Options

To create a report that displays time in microseconds (10^{-6} seconds) with a precision of three decimal places, select options by using name-value pairs.

```
report(myExecutionProfile, ...
    'Units','Seconds', ...
    'ScaleFactor','1e-06', ...
    'NumericFormat','%0.3f')
```

Create Report Comparing Against Baseline

To create a report that compares execution-time performance against a baseline, use the `baseline` option.

```
report(myExecutionProfile, ...
    'baseline', executionProfileBaseline)
```

Input Arguments

myExecutionProfile — Workspace variable created by `getCoderExecutionProfile`
workspace variable

The `myExecutionProfile` is a workspace variable that you create by using the `getCoderExecutionProfile` function.

Example: `myExecutionProfile`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `'Units', 'Seconds'`

Units — Time measurement unit

`'Seconds'` (default for SIL simulation on Windows) | `'Ticks'` (default for SIL simulation on non-Windows and PIL simulation unless overridden to `'Seconds'` by number of ticks per second in the target connectivity configuration)

Time measurements displayed in seconds or timer ticks.

Example: `'Units', 'Seconds'`

ScaleFactor — Scale factor for displayed measurements

`'1e-9'` (default) | character vector representation of a number that is a power of 10

To display measurements in microseconds, use a scale factor name-value pair `'ScaleFactor', '1e-6'`. The 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'`.

Example: `'ScaleFactor', '1e-9'`

NumericFormat — Numeric format for displayed measurements

`'%0.0f'` (default) | decimal convention utilized by the ANSI C fallback for ANSI C function `sprintf`

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'`.

Example: `'NumericFormat', '%0.0f'`

baseline — Baseline for performance comparison

`executionProfileBaseline`

The `executionProfileBaseline` is the workspace variable that contains baseline execution-time metrics.

Example: `'baseline', executionProfileBaseline`

Version History

Introduced in R2011b

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `Sections`

Topics

“Generate Execution Time Profile”

“Analyze Execution Time Data”

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)
```


Version History

Introduced in R2009a

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"](#)

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)
```

Version History

Introduced in R2009a

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”](#)

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)
```

Version History

Introduced in R2009a

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”

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.

Version History

Introduced in R2009a

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"](#)

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('libmwrrtiostreamtcpip.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 pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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.

Version History

Introduced in R2008b

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”

rtw.codegenObjectives.Objective

Custom code generation objective

Description

Use a code generation objective to specify a set of checks and parameters for the Code Generation Advisor to verify.

Creation

Syntax

```
rtw.codegenObjectives.Objective(newObjID)
rtw.codegenObjectives.Objective(newObjID, baseObjID)
```

Description

`rtw.codegenObjectives.Objective(newObjID)` creates a new code generation objective that has the specified identifier.

`rtw.codegenObjectives.Objective(newObjID, baseObjID)` creates a code generation objective based on the specified existing code generation objective. The existing objective must be registered in the Code Generation Advisor.

Input Arguments

newObjID — Identifier of new objective

character vector | string scalar

Identifier of new objective, specified as a character vector or string scalar.

baseObjID — Identifier of existing objective

character vector | string scalar

Identifier of existing objective that the new objective is based on, specified as a character vector or string scalar.

Object Functions

<code>setObjectiveName</code>	Specify objective name
<code>addCheck</code>	Add check to code generation objective
<code>addParam</code>	Add parameters to code generation objective
<code>excludeCheck</code>	Exclude check from code generation objective
<code>modifyInheritedParam</code>	Modify inherited parameter values in code generation objective
<code>removeInheritedCheck</code>	Remove inherited check from objective
<code>removeInheritedParam</code>	Remove inherited parameter from objective
<code>register</code>	Register objective in Code Generation Advisor

Examples

Create a Custom Code Generation Objective

Create a custom objective named `Reduced RAM Example` that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();
```

`end`

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

```
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, 'InlineParams', 'on');
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, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');

%Register the objective
register(obj);

end
```

Version History

Introduced in R2009a

See Also

Topics

“Create Custom Code Generation Objectives”

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
<code>getComponentPath</code>	<i>cmpPath</i> = <i>compArgs.getComponentPath</i> 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.
<code>getComponentCodePath</code>	<i>cmpCodePath</i> = <i>compArgs.getComponentCodePath</i> 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.
<code>getComponentCodeName</code>	<i>cmpCodeName</i> = <i>compArgs.getComponentCodeName</i> returns the component name used for code generation.
<code>getApplicationCodePath</code>	<i>appCodePath</i> = <i>compArgs.getApplicationCodePath</i> returns the folder path associated with the target application, for example, the path associated with the PIL application.
<code>getParam</code>	<i>settingOrParameterValue</i> = <i>compArgs.getParam(settingOrParameterName)</i> 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”

Version History

Introduced in R2008b

See Also

`rtw.connectivity.Config`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”

“Create PIL Target Connectivity Configuration for Simulink”

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);
```

- 5 Optionally, for stack usage profiling, use the `setStackDriver` method to specify a driver implementation that obtains stack usage data from the target hardware. For example, if you use `coder.profile.StackDriver` to create the object `stackUsageDriver`, insert the following line:

```
this.setStackDriver(stackUsageDriver)
```

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”

Version History

Introduced in R2008b

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 for MATLAB”

"Specify Hardware Timer for Simulink"

"Remove Instrumentation Overheads from Execution Time Measurements"

"Implement Driver to Obtain Stack Usage Data During PIL Simulation"

"Implement Driver to Obtain Stack Usage Data During PIL Execution"

"SIL and PIL Limitations"

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.

TargetHWDDeviceType — 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 `TargetHWDDeviceType` 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.Utls.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'};

% 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'};  
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”

Version History

Introduced in R2008b

See Also

`rtw.connectivity.Config`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”

“Create PIL Target Connectivity Configuration for Simulink”

“Register Code Replacement Library”

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
<code>getComponentArgs</code>	<code>componentArgs = obj.getComponentArgs</code> returns the <code>rtw.connectivity.ComponentArgs</code> object associated with the launcher object.
<code>setExe</code>	<code>setExe(exe)</code> specifies the application that runs on the target processor.
<code>getExe</code>	<code>exe=getExe()</code> returns the application that is running on the target processor.
<code>startApplication</code>	<i>obj.startApplication</i> 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 <code>setExe</code> method, which specifies the target application to run. To obtain this application, use the <code>getExe</code> method. For example: <pre>exe = getExe()</pre> The <code>startApplication</code> method resets the application to its initial state by ensuring that external and static (global) variables are zero-initialized.
<code>stopApplication</code>	<i>obj.stopApplication</i> is an abstract method that you must implement in a subclass. Called by MATLAB to stop execution of the target application.

Function	Description
<code>getApplicationStatus</code>	<i>obj</i>.<code>getApplicationStatus</code> is an optional method that you can implement in a subclass. Called by MATLAB or Simulink to detect the current status of the target application. The expected return values are: • <code>rtw.connectivity.LauncherApplicationStatus.UNKNOWN</code> • <code>rtw.connectivity.LauncherApplicationStatus.NOT_RUNNING</code> • <code>rtw.connectivity.LauncherApplicationStatus.RUNNING</code> If you do not implement the method, the default return value is <code>rtw.connectivity.LauncherApplicationStatus.UNKNOWN</code>.
<code>getBuilder</code>	<code>builder = obj.getBuilder</code> returns the <code>rtw.connectivity.Builder</code> 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”

Version History

Introduced in R2008b

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”

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_vc64.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”

Version History

Introduced in R2008b

See Also

`rtw.pil.RtIOStreamApplicationFramework` | `rtw.connectivity.ComponentArgs`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”

“Create PIL Target Connectivity Configuration for Simulink”

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
<code>setTimeoutRecvSecs</code>	<code>hostCommunicator.setTimeoutRecvSecs(<i>timeout</i>)</code> 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 <code>timeout</code> seconds.
<code>setInitCommsTimeout</code>	<code>hostCommunicator.setInitCommsTimeout(<i>timeout</i>)</code> 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”

Version History

Introduced in R2008b

See Also

`rtw.connectivity.ComponentArgs` | `rtw.connectivity.Launcher` | `rtiostream_wrapper`

Topics

“Create PIL Target Connectivity Configuration for MATLAB”

“Create PIL Target Connectivity Configuration for Simulink”

coder.profile.StackDriver

Specify driver to obtain stack usage data from target hardware

Description

Specify a driver implementation that obtains stack usage data from the target hardware during processor-in-the-loop (PIL) execution

Creation

`stackUsageDriver = coder.profile.StackDriver()` returns an object. Specify the driver implementation by providing values for these properties:

- `PtrDataType`
- `HeaderFile`
- `SourceFile`
- `IncludePaths`
- `DriverFunction`

Properties

PtrDataType — Pointer data type

'uint8' | 'uint16' | 'uint32' | 'uint64'

Data type of pointer that the driver function returns. If the target hardware is your development computer, use 'uint64'.

Example: `stackUsageDriver.PtrDataType = 'uint64';`

Data Types: char

HeaderFile — Header file

character vector

Name of the header file that must be included in the compilation of the driver function.

Example: `stackUsageDriver.HeaderFile = 'myHeaderFile.h';`

Data Types: char

SourceFile — Source file

character vector

Name of the source file for the driver function.

Example: `stackUsageDriver.SourceFile = 'mySourceFile.c';`

Data Types: char

IncludePaths — Include folder paths

cell array of character vectors

Paths to folders that must be included in the compilation of the driver function.

```
Example: stackUsageDriver.IncludePaths = {'path/To/myFolder1', ... 'path/To/myFolder2', ... 'path/To/myFolder3'};
```

Data Types: char

DriverFunction — Driver function

character vector

Name of the driver function.

```
Example: stackUsageDriver.DriverFunction = 'myDriverFunction';
```

Data Types: char

Examples**Obtain Stack Usage Data From Target Hardware.**

A PIL execution requires a driver implementation that obtains stack usage data from the target hardware.

When you set up PIL target connectivity, specify the driver through an `rtw.connectivity.Config` subclass. This code provides an example.

```
classdef overheadConnectivityConfig < rtw.connectivity.Config
    methods
        function this = customConnectivityConfig(componentArgs)

            % Create builder
            targetApplicationFramework = ...
                mypil.TargetApplicationFramework(componentArgs);
            builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
                targetApplicationFramework, '');

            % Create launcher
            launcher = mypil.Launcher(componentArgs, builder);

            % Set up communication
            hostCommunicator = rtw.connectivity.RtIOStreamHostCommunicator(...
                componentArgs, ...
                launcher, ...
                rtiostreamLibTCPIP);

            % Call super class constructor to register components
            this@rtw.connectivity.Config(componentArgs,...
                builder,...
                launcher,...
                hostCommunicator);

            % Specify driver implementation that obtains stack usage
            % data from the target hardware
            stackUsageDriver = coder.profile.StackDriver();
```



```

        stackUsageDriver.PtrDataType = 'uint64';
        stackUsageDriver.HeaderFile = 'myHeaderFile.h';
        stackUsageDriver.SourceFile = 'mySourceFile.c';
        stackUsageDriver.IncludePaths = {'path/To/myFolder1', ...
                                         'path/To/myFolder2', ...
                                         'path/To/myFolder3'};

        stackUsageDriver.DriverFunction = 'myDriverFunction';
        this.setStackDriver(stackUsageDriver);
    end
end
end

```

For more information about setting up PIL target connectivity, see:

- “Create PIL Target Connectivity Configuration for Simulink”
- “Create PIL Target Connectivity Configuration for MATLAB”

Version History

Introduced in R2022a

See Also

`rtw.connectivity.Config`

Topics

“View and Compare Stack Usage Metrics”

“Stack Usage Profiling for Code Generated From MATLAB Code”

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
<code>getComponentArgs</code>	<code>componentArgs = appFrameObj.getComponentArgs</code> returns the <code>rtw.connectivity.ComponentArgs</code> object associated with <code>appFrameObj</code> .
<code>getBuildInfo</code>	<code>buildInfo = appFrameObj.getBuildInfo</code> returns the <code>RTW.BuildInfo</code> object associated with <code>appFrameObj</code> .

Function	Description
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: <pre>appFrameObj.addPILMain('target');</pre> To specify a main function that is adapted for PIL on your development computer, enter: <pre>appFrameObj.addPILMain('host');</pre> Alternatively, you can specify your own main function: <pre>componentArgs = appFrameObj.getComponentArgs; buildInfo = appFrameObj.getBuildInfo; buildInfo.addSourcePaths(pathToMyMainC); buildInfo.addSourceFiles(myMainC);</pre>

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”

Version History

Introduced in R2008b

See Also

rtw.connectivity.ComponentArgs | rtiostream_wrapper

Topics

“Create PIL Target Connectivity Configuration for MATLAB”

“Create PIL Target Connectivity Configuration for Simulink”

“Build Information Object”

run

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 `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...	<i>cgv.CGV.run()</i> executes callback function...
HeaderReportFcn	addHeaderReportFcn	Before executing input data in <i>cgv.CGV</i>
PreExecReportFcn	addPreExecReportFcn	Before executing each input data file in <i>cgv.CGV</i>
PreExecFcn	addPreExecFcn	Before executing each input data file in <i>cgv.CGV</i>
PostExecReportFcn	addPostExecReportFcn	After executing each input data file in <i>cgv.CGV</i>

Callback function	Add to object using...	<code>cgv.CGV.run()</code> executes callback function...
PostExecFcn	addPostExecFcn	After executing each input data file in <code>cgv.CGV</code>
TrailerReportFcn	addTrailerReportFcn	After the input data is executed in <code>cgv.CGV</code>

See Also

Topics

“Verify Numerical Equivalence with CGV”

setMode

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
<code>sim</code> or <code>normal</code> (default)	Mode of execution is normal simulation.
<code>sil</code>	Mode of execution is SIL.
<code>pil</code>	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 = 'myCGVModel';  
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');  
cgvObj1.run();  
cgvObj2 = cgvObj1.copySetup()  
cgvObj2.setMode('sil');  
cgvObj2.run();
```

See Also

[run](#) | [copySetup](#)

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'`

Version History

Introduced in R2010a

See Also

`enableCPP` | `registerCPPFunctionEntry`

Topics

“Define Code Replacement Library Optimizations”

setObjectiveName

Namespace: rtw.codegenObjectives

Specify objective name

Syntax

```
setObjectiveName(objective, name)
```

Description

setObjectiveName(objective, name) specifies the name for the code generation objective. The Configuration Set Objectives dialog box displays the name of the objective.

Examples

Create a Custom Code Generation Objective

Create a custom objective named `Reduced RAM Example` that runs checks and verifies parameter values to confirm that the model is configured to reduce the RAM used by the generated code.

Create a file `sl_customization.m` to contain a callback function that creates the custom objective.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

Create and configure the objective in the `addObjectives` function. Set the name of the objective and add checks and parameters to verify. Then register the objective in the Code Generation Advisor.

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

```
%Register the objective  
register(obj);  
  
end
```

Input Arguments

objective — Code generation objective

`rtw.codegenObjectives.Objective` object

Code generation objective, specified as a `rtw.codegenObjectives.Objective` object.

Example:

name — Name of objective

character vector | string scalar

Name of objective, specified as a character vector or string scalar.

Example:

Version History

Introduced in R2009a

See Also

Topics

“Create Custom Code Generation Objectives”

setOutputDir

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

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

Version History

Introduced in R2008a

See Also

Topics

“Reserved Identifiers and Code Replacement”

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 pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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_LOOKUP'` | `'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`, `sincos`, and `atan2`. 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	RTW_CORDIC	Match the CORDIC approximation method
cos	RTW_LOOKUP	Match the Lookup approximation method
sincos	RTW_DEFAULT	Match the default approximation method, None
atan2	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.

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

Version History

Introduced in R2007b

See Also

[addAdditionalHeaderFile](#) | [addAdditionalIncludePath](#) | [addAdditionalLinkObj](#) |
[addAdditionalLinkObjPath](#) | [addAdditionalSourceFile](#) | [addAdditionalSourcePath](#)

Topics

"Define Code Replacement Library Optimizations"

"Code You Can Replace from MATLAB Code"

"Code You Can Replace From Simulink Models"

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
<i>hEntry</i> = RTW.TfLCOperationEntry;	Supports operator replacement.
<i>hEntry</i> = RTW.TfLCOperationEntryGenerator;	Provides parameters for fixed-point addition and subtraction that are not available in RTW.TfLCOperationEntry (SlopesMustBeTheSame and MustHaveZeroNetBias).
<i>hEntry</i> = RTW.TfLCOperationEntryGenerator_NetSlope;	Provides net slope parameters for fixed-point multiplication and division that are not available in RTW.TfLCOperationEntry (NetSlopeAdjustmentFactor and NetFixedExponent).
<i>hEntry</i> = RTW.TfLBlasEntryGenerator;	Supports replacement of nonscalar operators with MathWorks BLAS functions.

Class Instantiation	Support
<code>hEntry = RTW.TfllCBlasEntryGenerator;</code>	Supports replacement of nonscalar operators with ANSI/ISO® C BLAS functions.
<code>hEntry = MyCustomOperationEntry;</code> (where <i>MyCustomOperationEntry</i> is a class derived from <i>RTW.TfllCOperationEntry</i>)	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.TfllCOperationEntryGenerator` rather than `hEntry = RTW.TfllCOperationEntry`. If you want to use *NetSlopeAdjustmentFactor* and *NetFixedExponent*, instantiate your table entry by using `hEntry = RTW.TfllCOperationEntryGenerator_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 pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where *Name* is the argument name and *Value* is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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_FUNC` 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.

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

Version History

Introduced in R2007b

See Also

[addAdditionalHeaderFile](#) | [addAdditionalIncludePath](#) | [addAdditionalLinkObj](#) | [addAdditionalLinkObjPath](#) | [addAdditionalSourceFile](#) | [addAdditionalSourceFile](#)

Topics

“Define Code Replacement Library Optimizations”
 “Scalar Operator Code Replacement”
 “Replace Addition and Subtraction Operator Code”
 “Small Matrix Operation to Processor Code Replacement”
 “Code You Can Replace from MATLAB Code”
 “Code You Can Replace From Simulink Models”

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 pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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.

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

Version History

Introduced in R2013a

See Also

[addAdditionalHeaderFile](#) | [addAdditionalIncludePath](#) | [addAdditionalLinkObj](#) | [addAdditionalLinkObjPath](#) | [addAdditionalSourceFile](#) | [addAdditionalSourcePath](#)

Topics

"Define Code Replacement Library Optimizations"
 "Code You Can Replace from MATLAB Code"
 "Code You Can Replace From Simulink Models"
 "Mutex and Semaphore Functions"

`coder.MATLABCodeTemplate.setTokenValue`

Class: `coder.MATLABCodeTemplate`

Namespace: `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.getTokenValue` |
`coder.MATLABCodeTemplate.getcurrenttokens` |
`coder.MATLABCodeTemplate.emitSection`

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.TfLFunctionEntry;
```

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

```
setTfLFunctionEntryParameters(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]
    ApplyFullPrecisionForLinearInterpolation: [1x1 coder.algorithm.parameter.ApplyFullPrecisionForLinearInterpolation]
    RemoveProtectionInput: [1x1 coder.algorithm.parameter.RemoveProtectionInput]
    SaturateOnIntegerOverflow: [1x1 coder.algorithm.parameter.SaturateOnIntegerOverflow]
    SupportTunableTableSize: [1x1 coder.algorithm.parameter.SupportTunableTableSize]
    BPPower2Spacing: [1x1 coder.algorithm.parameter.BPPower2Spacing]
    BeginIndexSearchUsingPreviousIndexResult: [1x1 coder.algorithm.parameter.BeginIndexSearchUsingPreviousIndexResult]
```

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`.

Version History

Introduced in R2015a

See Also

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

Topics

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

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. Before calling other default mapping functions, 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 `ConfigurationInterface`, create the environment for configuring data and functions for code generation.

```
openExample('ConfigurationInterface')
coder.mapping.create('ConfigurationInterface');
```

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

Version History

Introduced in R2018a

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 Data Elements and Functions”

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`, `ModelParameters`, and `InternalData` by using calls to `coder.mapping.defaults.allowedProperties`.

Load the model.

```
openExample('ConfigurationInterface');
```

Get a list of the properties for the model default data category `Inports`.

```
coder.mapping.defaults.allowedProperties('ConfigurationInterface',...
    'Inports')
```

ans =

3×1 cell array

```
{'StorageClass'      }
{'HeaderFile'        }
{'PreserveDimensions'}
```

Get a list of the properties for the model default data category `Outports`.

```
coder.mapping.defaults.allowedProperties('ConfigurationInterface',...
    'Outports')
```

ans =

5×1 cell array

```
{'StorageClass'      }
{'HeaderFile'        }
```

```
{'DefinitionFile'    }  
{'Owner'            }  
{'PreserveDimensions'}
```

Get a list of the properties for the model default data category `ModelParameters`.

```
coder.mapping.defaults.allowedProperties('ConfigurationInterface',...  
    'ModelParameters')
```

ans =

5×1 cell array

```
{'StorageClass'      }  
{'HeaderFile'        }  
{'DefinitionFile'    }  
{'Owner'            }  
{'PreserveDimensions'}
```

Get a list of the properties for the model default data category `InternalData`.

```
coder.mapping.defaults.allowedProperties('ConfigurationInterface',...  
    '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`.

Load the model.

```
openExample('ConfigurationInterface');
```

Get a list of the properties for the model default function category `InitializeTerminate`.

```
catData = coder.mapping.defaults.allowedProperties('ConfigurationInterface',...  
    'InitializeTerminate');  
catData
```

catData =

2×1 cell array

```
{'FunctionCustomizationTemplate'}  
{'MemorySection'}
```

Get a list of the properties for the model default function category `Execution`.

```
catFunctions = coder.mapping.defaults.allowedProperties('ConfigurationInterface',...  
    'Execution');  
catFunctions
```

catFunctions =

```
2×1 cell array
```

```
{'FunctionCustomizationTemplate'}
{'MemorySection'}
```

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.

Version History

Introduced in R2018a

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 Data Elements and Functions”

`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 Model Parameters

Get the list of values that you can specify for property `StorageClass` for model default data category `ModelParameters` by calling `coder.mapping.defaults.allowedValues`.

```
lclparam_scs = coder.mapping.defaults.allowedValues('ConfigurationInterface', 'ModelParameters',...  
    'StorageClass')  
lclparam_scs  
  
lclparam_scs =  
  
    14x1 cell array  
  
    {'Default'      }  
    {'ExportedGlobal'}  
    {'ImportedExtern'}  
    {'ImportedExternPointer'}  
    {'Const'        }  
    {'Volatile'      }  
    {'ConstVolatile'}  
    {'Define'        }  
    {'ImportedDefine'}  
    {'ExportToFile'  }  
    {'ImportFromFile'}  
    {'FileScope'     }  
    {'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: `'ModelParameters'`

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.

Version History

Introduced in R2018a

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 Data Elements and Functions”

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 =

    1×9 cell array

    Columns 1 through 4

        {'Inports'}      {'Outports'}      {'ModelParameters'}      {'ModelParameterA...'}

    Columns 5 through 8

        {'ExternalParamet...'}      {'SharedLocalData...'}      {'GlobalDataStores'}      {'InternalData'}

    Column 9

        {'Constants'}
```

Output Arguments

categories — Names of data categories

cell array

Cell array of names for default mapping data categories.

Version History

Introduced in R2018a

See Also

`coder.mapping.defaults.allowedProperties` | `coder.mapping.defaults.allowedValues`
| `coder.mapping.defaults.functionCategories` | `coder.mapping.defaults.get` |
`coder.mapping.defaults.set`

Topics

“Configure Default C Code Generation for Categories of Data Elements and Functions”

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.

Version History

Introduced in R2018a

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 Data Elements and Functions”

`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 `ConfigurationInterface`, 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 =

    1×9 cell array

    Columns 1 through 4

        {'Inports'}    {'Outports'}    {'ModelParameters'}    {'ModelParameterA...'}

    Columns 5 through 8

        {'ExternalParamet...'}    {'SharedLocalData...'}    {'GlobalDataStores'}    {'InternalData'}

    Column 9

        {'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('ConfigurationInterface', '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('ConfigurationInterface', 'Inports', 'StorageClass')

ans =

    'ImportFromFile'
```

Return Memory Section Setting for Model Execution Entry-Point Functions

For model `ConfigurationInterface`, 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 =

    1x3 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('ConfigurationInterface', 'Execution')

ans =

    1x1 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('ConfigurationInterface', '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: 'ModelParameters'

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.

Version History

Introduced in R2018a

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 Data Elements and Functions”

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 `ConfigurationInterface`, 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`.

```
openExample('ConfigurationInterface')
coder.mapping.defaults.dataCategories()

ans =

    1×9 cell array

    Columns 1 through 4

    {'Inports'}    {'Outports'}    {'ModelParameters'}    {'ModelParameterA...'}

    Columns 5 through 8

    {'ExternalParamet...'}    {'SharedLocalData...'}    {'GlobalDataStores'}    {'InternalData'}

    Column 9

    {'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('ConfigurationInterface', 'Outports')
```

```
ans =  
  
5x1 cell array  
  
{'StorageClass'      }  
{'HeaderFile'       }  
{'DefinitionFile'    }  
{'Owner'            }  
{'PreserveDimensions'}
```

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('ConfigurationInterface', 'Outports', 'StorageClass')
```

```
ans =  
  
10x1 cell array  
  
{'Default'           }  
{'ExportedGlobal'    }  
{'ImportedExtern'     }  
{'ImportedExternPointer' }  
{'Volatile'          }  
{'ExportToFile'       }  
{'ImportFromFile'     }  
{'Localizable'        }  
{'GetSet'             }  
{'MultiInstance'     }
```

```
coder.mapping.defaults.allowedValues('ConfigurationInterface', 'Outports', 'HeaderFile')
```

```
ans =  
  
0x1 empty cell array
```

```
coder.mapping.defaults.allowedValues('ConfigurationInterface', 'Outports', 'DefinitionFile')
```

```
ans =  
  
0x1 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('ConfigurationInterface', 'Outports', ...  
    'Storageclass', 'ExportToFile', ...  
    'HeaderFile', 'exSysOut.h', ...  
    'DefinitionFile', 'exSysOut.c')
```

Configure Default Location in Memory for Storing Code Generated for Model Internal Data

For model `ConfigurationInterface`, 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`.

```
openExample('ConfigurationInterface')  
coder.mapping.defaults.dataCategories()
```

```
ans =
    1×9 cell array

    Columns 1 through 4
        {'Inports'}    {'Outports'}    {'ModelParameters'}    {'ModelParameterA...'}

    Columns 5 through 8
        {'ExternalParamet...'}    {'SharedLocalData...'}    {'GlobalDataStores'}    {'InternalData'}

    Column 9
        {'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('ConfigurationInterface', '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('ConfigurationInterface', 'InternalData', 'MemorySection')
```

```
ans =
    2×1 cell array

    {'None' }
    {'MemVolatile'}
```

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('ConfigurationInterface', '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: `'ModelParameters'`

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'`

Version History

Introduced in R2018a

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 Data Elements and Functions”

getDeploymentType

Get deployment type of model

Syntax

```
depType = getDeploymentType(coderMapObj)
```

Description

`depType = getDeploymentType(coderMapObj)` returns the deployment type of the model specified by code mappings object `coderMapObj`. The deployment type controls the code interfaces at different levels of a model hierarchy. The code generator uses the deployment type to:

- Enforce peer and nesting rules in the model hierarchy.
- Map model elements to code interface definitions.
- Generate code that uses the appropriate interface to connect to other parts of the hierarchy.

Examples

Get Deployment Type of Model

The deployment type of a model controls the code interfaces at different levels of a model hierarchy.

To access the deployment type of a model, use the `getDeploymentType` function.

```
openExample("CounterModel.slx");
cm = coder.mapping.utils.create("CounterModel");
depType = getDeploymentType(cm)
```

```
depType =
    'Automatic'
```

The default deployment type of a model is `Automatic`.

To specify the deployment type of a model, use the `setDeploymentType` function.

```
setDeploymentType(cm, "Subcomponent")
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings), returned by a call to function `coder.mapping.api.get`.

Output Arguments

depType — Deployment type of the model

Automatic | Component | Subcomponent

The deployment type of the model code mappings object, returned as **Automatic**, **Component**, or **Subcomponent**.

Deployment Type	Description
Automatic	Embedded Coder determines the deployment type based on the model hierarchy context
Component	The top model that generates code. The model generates a standalone algorithm. The component code exposes its interface to other components in the system. Component deployment type is only valid for models with a service interface code configuration.
Subcomponent	A model reference that a component model uses. The generated code entry points are symbolically scoped to the parent component.

Version History

Introduced in R2022a

See Also

`coder.mapping.api.CodeMapping` | `setDeploymentType`

Topics

“Generate Code for Model Reference Hierarchy”

setDeploymentType

Set deployment type of model

Syntax

```
setDeploymentType(coderMapObj, depType)
```

Description

`setDeploymentType(coderMapObj, depType)` sets the deployment type of the model specified by code mappings object `coderMapObj` to the type specified by `depType`. The deployment type controls the code interfaces at different levels of a model hierarchy. The code generator uses the deployment type to:

- Enforce peer and nesting rules in the model hierarchy.
- Map model elements to code interface definitions.
- Generate code that uses the appropriate interface to connect to other parts of the hierarchy.

Examples

Set Deployment Type of Model

The deployment type of a model controls the code interfaces at different levels of a model hierarchy.

To access the deployment type of a model, use the `getDeploymentType` function.

```
openExample("CounterModel.slx");
cm = coder.mapping.utils.create("CounterModel");
depType = getDeploymentType(cm)
```

```
depType =
    'Automatic'
```

The default deployment type of a model is `Automatic`.

To specify the deployment type of a model, use the `setDeploymentType` function.

```
setDeploymentType(cm, "Subcomponent")
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings), returned by a call to function `coder.mapping.api.get`.

depType — Deployment type of the model

Automatic (default) | Component | Subcomponent

The deployment type of the model, specified as Automatic, Component, or Subcomponent.

Deployment Type	Description
Automatic	Embedded Coder determines the deployment type based on the model hierarchy context
Component	The top model that generates code. The model generates a standalone algorithm. The component code exposes its interface to other components in the system. Component deployment type is only valid for models with a service interface code configuration.
Subcomponent	A model reference that a component model uses. The generated code entry points are symbolically scoped to the parent component.

Data Types: char

Version History

Introduced in R2022a

See Also

`coder.mapping.api.CodeMapping` | `getDeploymentType`

Topics

“Generate Code for Model Reference Hierarchy”

setDataTransfer

Namespace: `coder.mapping.api`

Set code configuration in code mappings for signal representing data transfer

Syntax

```
setDataTransfer(coderMapObj,portHandle,DataTransferService=
dataTransferService)
```

Description

`setDataTransfer(coderMapObj,portHandle,DataTransferService=dataTransferService)` sets the data transfer service of the signal corresponding to block output port, `portHandle`.

To configure the data transfer service, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model.

Examples

Configure the Data Transfer Service for a Signal

This example model references a shared coder dictionary that defines the default data transfer services for signals representing data transfers. To configure the data transfer to use something other than the default service, use the `getDataTransfer` and `setDataTransfer` functions.

Open the component model, `ComponentDeploymentFcn`.

```
openExample("ComponentDeploymentFcn");
```

Get the code mappings for the model using the `coder.mapping.api.get` function.

```
cm = coder.mapping.api.get("ComponentDeploymentFcn");
```

In this model, the signal connecting two Function-Call Subsystem blocks, `Accumulator` and `Integrator` represents a data transfer. Get the current data transfer service configuration using the `getDataTransfer` function.

```
ph = get_param("ComponentDeploymentFcn/Integrator","PortHandles");
dataTransServ = getDataTransfer(cm,ph.Outport,"DataTransferService")
```

```
dataTransServ =
```

```
    'Dictionary default'
```

The signal representing the data transfer is configured to use the default data transfer service defined in the shared coder dictionary.

To configure the signal to communicate data with other functions immediately during function execution instead of before or after execution as specified in the dictionary, use the

setDataTransfer function and set the DataTransferService property to DataTransferDuringExe.

```
setInport(cm,ph.Output,"DataTransferService","DataTransferDuringExe")
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

portHandle — Output port handle of signal source block

port handle

Port handle for block output signal representing a data transfer. In a component model, signal lines connecting communicating functions represent data transfers in the generated code. Communicating functions are modeled using Function-Call Subsystem blocks that are aperiodic or periodic functions.

dataTransferService — Name of data transfer service

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a data transfer service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify "Dictionary default".

In the Embedded Coder Dictionary, you can configure the data transfer services to one of the following:

- *Outside execution*: The platform communicates data with other functions outside (before and after) function execution.
- *During execution*: The platform service communicates data with other functions immediately during function execution.

For more information on data communication methods, see "Data Communication Methods".

Data Types: char | string

Version History

Introduced in R2022b

See Also

`coder.mapping.api.CodeMapping` | `getDataTransfer`

Topics

"Data Communication Methods"

"Configure Data Transfer Service Interfaces for Data Transfer Signals"

getDataTransfer

Namespace: `coder.mapping.api`

Get code configuration from code mappings for signal representing data transfer

Syntax

```
dataTransferService = getDataTransfer(coderMapObj,  
portHandle,"DataTransferService")
```

Description

`dataTransferService = getDataTransfer(coderMapObj, portHandle, "DataTransferService")` returns the data transfer service for the signal corresponding to block output port, `portHandle`.

To configure the data transfer service, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model.

Examples

Configure the Data Transfer Service for a Signal

This example model references a shared coder dictionary that defines the default data transfer services for signals representing data transfers. To configure the data transfer to use something other than the default service, use the `getDataTransfer` and `setDataTransfer` functions.

Open the component model, `ComponentDeploymentFcn`.

```
openExample("ComponentDeploymentFcn");
```

Get the code mappings for the model using the `coder.mapping.api.get` function.

```
cm = coder.mapping.api.get("ComponentDeploymentFcn");
```

In this model, the signal connecting two Function-Call Subsystem blocks, `Accumulator` and `Integrator` represents a data transfer. Get the current data transfer service configuration using the `getDataTransfer` function.

```
ph = get_param("ComponentDeploymentFcn/Integrator", "PortHandles");  
dataTransServ = getDataTransfer(cm, ph.Outputport, "DataTransferService")
```

```
dataTransServ =
```

```
    'Dictionary default'
```

The signal representing the data transfer is configured to use the default data transfer service defined in the shared coder dictionary.

To configure the signal to communicate data with other functions immediately during function execution instead of before or after execution as specified in the dictionary, use the

setDataTransfer function and set the DataTransferService property to DataTransferDuringExe.

```
setInport(cm,ph.Outputport,"DataTransferService","DataTransferDuringExe")
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

portHandle — Output port handle of signal source block

port handle

Port handle for block output signal representing a data transfer. In a component model, signal lines connecting communicating functions represent data transfers in the generated code. Communicating functions are modeled using Function-Call Subsystem blocks that are aperiodic or periodic functions.

Output Arguments

dataTransferService — Data transfer service of the signal

character vector

Name of the data transfer service of the specified signal.

Version History

Introduced in R2022b

See Also

`coder.mapping.api.CodeMapping` | `setDataTransfer`

Topics

“Data Communication Methods”

“Configure Data Transfer Service Interfaces for Data Transfer Signals”

coder.mapping.api.CodeMappingCPP

Model data and interface configuration for C++ code generation

Description

A code mappings object and related functions enable you to configure C++ code generation for data and functions of a Simulink model. For model data elements, code mappings associate data elements with configurations that consist of a storage class and storage class properties. For functions, code mappings associate entry-point functions with configurations that consist of a function customization template. Reduce the effort of preparing a model for C++ code generation by specifying default configurations for categories of data elements and functions across a model. Override default configurations by configuring data elements or functions individually. For smaller models, you can choose to configure each data element and function individually.

Creation

When you select the **Embedded Coder** app from the Apps tab in the Simulink Editor, the app creates a `coder.mapping.api.CodeMappingCPP` object if code mappings do not already exist. The app creates code mappings based on code customization settings stored in the model active configuration set object. The configuration set object can specify memory sections for data and functions.

Access a `coder.mapping.api.CodeMappingCPP` object programmatically by using the `coder.mapping.utils.create` or `coder.mapping.api.get` functions.

Object Functions

<code>find</code>	Get model elements for the category of model code mappings
<code>getClassName</code>	Get class name of model
<code>setClassName</code>	Set class name of model
<code>getClassNamespace</code>	Get class namespace for a model
<code>setClassNamespace</code>	Set class namespace of model
<code>getData</code>	Get code mapping configuration for model data
<code>setData</code>	Configure model data for C++ code generation
<code>getFunction</code>	Get code configuration from code mappings for model function
<code>setFunction</code>	Set code mapping information for model function

Examples

Create Environment to Configure C++ Code Mappings for Model

Open the model `CppClassRateBased`, and create the environment for configuring model data and functions for code generation by using the `coder.mapping.utils.create` function.

```
open_system('CppClassRateBased')
coder.mapping.utils.create('CppClassRateBased');
```

After calling `coder.mapping.utils.create`, use calls to other functions listed under Object Functions to configure aspects of code generation for model interface elements.

Version History

Introduced in R2021a

See Also

`coder.mapping.utils.create` | `coder.mapping.api.get`

Topics

“Interactively Configure C++ Interface”

“Programmatically Configure C++ Interface”

getClassName

Get class name of model

Syntax

```
name = getClassName(myCPPMappingObj)
```

Description

`name = getClassName(myCPPMappingObj)` returns the class name of the model.

Examples

Get Class Name of Model

Open the model. To access the CodeMappingCPP object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('CppClassRateBased');
cm = coder.mapping.api.get('CppClassRateBased');
```

Use the `getClassName` function to get the class name of the model.

If you did not specify a class name for the model, the `getClassName` function returns an empty character vector and the class name in the generated code uses the model name as the default class name.

```
name = getClassName(cm)
```

```
name =
'multi_rate'
```

Specify a class name for the model by using the `setClassName` function.

```
setClassName(cm, 'myClassName');
```

The `getClassName` function now returns the specified class name.

```
name = getClassName(cm)
```

```
name =
'myClassName'
```

Input Arguments

myCPPMappingObj — C++ code mapping object

CodeMappingCPP object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

Output Arguments

name — Class name of model

character vector

Class name of model, returned as a character vector. If you do not specify a class name, the class name in the generated code uses the name of the model as the default class name.

Version History

Introduced in R2021a

See Also

`setClassName` | `getClassNamespace` | `setClassNamespace` | `getData` | `setData` | `getFunction` | `setFunction`

Topics

“Interactively Configure C++ Interface”

“Programmatically Configure C++ Interface”

find

Get model elements for the category of model code mappings

Syntax

```
modelElementsFound = find(myCPPMappingObj,category)
modelElementsFound = find(myCPPMappingObj,category,'MethodName', methodName)
```

Description

`modelElementsFound = find(myCPPMappingObj,category)` returns the elements in the model code mappings of the specified category as an array of objects.

`modelElementsFound = find(myCPPMappingObj,category,'MethodName', methodName)` returns the model functions in the model code mappings with the specified method name.

Examples

Find All Model Functions with an Unset Method Name

You can use the `find` function to find all model elements of a specific category in the code mappings.

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('CppClassWorkflowKeyIgnition');
cm = coder.mapping.api.get('CppClassWorkflowKeyIgnition');
```

Find all functions with an unset method name by using the `find` function.

```
unsetMethods = find(cm, 'Functions', 'MethodName', '')

unsetMethods = 1x4 string
    "Initialize"    "Terminate"    "Periodic:D1"    "Periodic:D2"
```

To specify a method name for the functions, use the `setFunction` function.

```
setFunction(cm, unsetMethods, 'MethodName', 'my_$N')
```

Input Arguments

myCPPMappingObj — C++ code mapping object

`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

category — Model element category

Functions | ExportedFunctions | PartitionFunctions | PeriodicFunctions |
ResetFunctions | SimulinkFunctions

Category of model elements to search for in the model code mappings.

Example: 'ResetFunctions'

methodName — Name of entry-point function

character vector | string scalar

Name of an entry-point function generated for a model.

Example: 'my_\$N'

Data Types: char | string

Output Arguments

modelElementsFound — Model elements found

array | string vector

Model elements found, returned as an array or string vector of functions. Each object identifies a model element of the specified category. If you specify additional search criteria, the array or string vector includes objects for model elements of the specified category that meet the additional search criteria.

Version History

Introduced in R2021a

See Also

getClassName | setClassName | getClassNamespace | setClassNamespace | getData |
setData | getFunction | setFunction

Topics

“Interactively Configure C++ Interface”
“Programmatically Configure C++ Interface”

setClassName

Set class name of model

Syntax

```
setClassName(myCPPMappingObj, name)
```

Description

setClassName(myCPPMappingObj, name) sets the class name of the model in the generated code.

Examples

Set Class Name of Model

Open the model. To access the CodeMappingCPP object associated with the model, use the coder.mapping.api.get function.

```
open_system('CppClassRateBased');
cm = coder.mapping.api.get('CppClassRateBased');
```

To access the class name of the model, use the getClassName function. If you did not specify a class name for the model, the getClassName function returns an empty character vector and the class name in the generated code uses the model name as the default class name.

```
name = getClassName(cm)
```

```
name =
'multi_rate'
```

Specify a class name for the model by using the setClassName function.

```
setClassName(cm, 'myClassName');
```

The getClassName function now returns the specified class name.

```
name = getClassName(cm)
```

```
name =
'myClassName'
```

Input Arguments

myCPPMappingObj — C++ code mapping object

CodeMappingCPP object

C++ code mapping object, returned by a call to either the coder.mapping.utils.create function or the coder.mapping.api.get function.

name — Class name of model

character vector

Class name of model in the generated code, specified as a character vector. If you do not specify a class name, the class name in the generated code is set to the model name.

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`find` | `getClassName` | `getClassNamespace` | `setClassNamespace` | `getData` | `setData` | `getFunction` | `setFunction`

Topics

“Interactively Configure C++ Interface”

“Programmatically Configure C++ Interface”

getClassNamespace

Get class namespace for a model

Syntax

```
namespace = getClassNamespace(myCPPMappingObj)
```

Description

`namespace = getClassNamespace(myCPPMappingObj)` returns the class namespace specified for the model. Class namespaces can help to prevent name conflicts in large projects.

Examples

Access Class Namespace for Model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('CppClassRateBased');
cm = coder.mapping.api.get('CppClassRateBased');
```

Use the `getClassNamespace` function to access the namespace of the model.

If you did not specify a namespace for the model, the `getClassNamespace` function returns an empty character vector.

```
name = getClassNamespace(cm)
```

```
name =
'example_ns'
```

Specify a namespace for the model by using the `setClassNamespace` function.

```
setClassNamespace(cm, 'myClassNamespace');
```

The `getClassNamespace` function now returns the specified class namespace.

```
name = getClassNamespace(cm)
```

```
name =
'myClassNamespace'
```

Input Arguments

myCPPMappingObj — C++ code mapping object

`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

Output Arguments

namespace — Class namespace of model

character vector

Class namespace of model, returned as a character vector. If you did not specify a namespace for the model, the `getClassNamespace` function returns an empty character vector.

Version History

Introduced in R2021a

See Also

`setClassNamespace` | `getClassName` | `setClassName` | `getData` | `setData` | `getFunction` | `setFunction` | `find`

Topics

“Interactively Configure C++ Interface”

“Programmatically Configure C++ Interface”

setClassNamespace

Set class namespace of model

Syntax

```
setClassNamespace(myCPMappingObj, namespace)
```

Description

`setClassNamespace(myCPMappingObj, namespace)` sets the class namespace of the model in the generated code. Control the scope of the generated code by specifying a namespace for the generated class. In systems that use a model hierarchy, you can specify a different namespace for each model in the hierarchy.

Examples

Set Class Namespace for Model

Specify a class namespace for a model and generate C++ code.

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('CppClassRateBased');
cm = coder.mapping.api.get('CppClassRateBased');
```

To specify a namespace for the model in the generated code, use the `setClassNamespace` function.

```
setClassNamespace(cm, 'myClassNamespace');
```

To configure a nested namespace, use the scope resolution operator `::` to specify scope.

```
setClassNamespace(cm, 'myNestedClassNamespace1::ns2::ns3');
```

Input Arguments

myCPMappingObj — C++ code mapping object

`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

namespace — Class namespace of model

character vector

Class namespace of model in the generated code, specified as a character vector. If you do not specify a class namespace, the model class is generated at the global scope. To configure a nested namespace, use the scope resolution operator `::` in the form `ns1::ns2::ns3`.

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`find` | `getClassName` | `setClassName` | `getClassNamespace` | `getData` | `setData` | `getFunction` | `setFunction`

Topics

“Interactively Configure C++ Interface”

“Programmatically Configure C++ Interface”

getData

Get code mapping configuration for model data

Syntax

```
value = getData(myCPPMappingObj, category, property)
```

Description

`value = getData(myCPPMappingObj, category, property)` returns the code mapping information for the property specified by `property` and the model data category specified by `category`.

Examples

Get Data Visibility of Model Parameters

Get the data visibility of model parameters.

Open the model. Use the `coder.mapping.api.get` function to access the `CodeMappingCPP` object associated with the model.

```
open_system('CppClassRateBased');
cm = coder.mapping.api.get('CppClassRateBased');
```

To view the data visibility of the model parameters, use the `getData` function.

```
value = getData(cm, 'ModelParameters', 'DataVisibility')

value =
'public'
```

To configure the data visibility, specify the `'DataVisibility'` parameter by using the `setData` function.

```
setData(cm, 'ModelParameters', 'DataVisibility', 'private');
```

Input Arguments

myCPPMappingObj — C++ code mapping object

`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

category — Category of model data

`'Inports' | 'Outports' | 'ModelParameters' | 'ModelParameterArguments' | 'InternalData'`

Category of model data to access, specified as one of these categories.

Category	Description
'Inports'	Root-level input ports of a model, such as Inport and In Bus Element blocks.
'Outports'	Root-level output ports of a model, such as Outport and Out Bus Element blocks.
'ModelParameters'	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
'ModelParameterArguments'	Parameters in the model workspace configured as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value.
'InternalData'	Data elements that are internal to a model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

Data Types: char | string

property — Property of model data

'MemberAccessMethod' | 'DataVisibility' | 'DataAccess'

Property of model data to access, specified as 'MemberAccessMethod', 'DataVisibility', or 'DataAccess'.

The MemberAccessMethod property specifies how the methods, if any, are generated for the data elements.

The DataVisibility property specifies the visibility (private, public, or protected) of the data category in the generated code.

The DataAccess property specifies whether model elements are stored by value ('Direct') or pointer ('Pointer') in the generated code. Data access is configurable for 'Inports', 'Outports', and 'ModelParameterArguments' categories:

- 'Inports' whose member access method is set to 'Structure-based method' or 'Inlined structure-based method'.
- 'Outports' whose member access method is set to 'Structure-based method' or 'Inlined structure-based method'.
- 'ModelParameterArguments' whose data visibility is set to 'private' (not 'Individual Arguments').

Data Types: char | string

Output Arguments

value — Code mapping property value of category

character vector

The code mapping property value of the specified category, returned as a character vector.

Version History

Introduced in R2021a

See Also

find | getClassName | setClassName | getClassNamespace | setClassNamespace | setData |
getFunction | setFunction

Topics

"Interactively Configure C++ Interface"

"Programmatically Configure C++ Interface"

setData

Configure model data for C++ code generation

Syntax

```
setData(myCPMappingObj, category, Name,Value)
```

Description

`setData(myCPMappingObj, category, Name,Value)` configures code mapping information for the model data specified by category.

Examples

Configure Data Visibility of Model Parameters

Configure the data visibility of model parameters to be private.

Open the model. Use the `coder.mapping.api.get` function to access the `CodeMappingCPP` object associated with the model.

```
open_system('CppClassRateBased');  
cm = coder.mapping.api.get('CppClassRateBased');
```

To view the data visibility of the model parameters, use the `getData` function.

```
value = getData(cm, 'ModelParameters', 'DataVisibility')  
  
value =  
'public'
```

To configure the data visibility, specify the `'DataVisibility'` parameter using the `setData` function.

```
setData(cm, 'ModelParameters', 'DataVisibility', 'private');
```

Input Arguments

myCPMappingObj — C++ code mapping object

`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

category — Category of model data

`'Imports' | 'Outputs' | 'ModelParameters' | 'ModelParameterArguments' | 'InternalData'`

Category of model data to configure, specified as one of these categories.

Category	Description
'Inports'	Root-level input ports of a model, such as Inport and In Bus Element blocks.
'Outports'	Root-level output ports of a model, such as Outport and Out Bus Element blocks.
'ModelParameters'	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.
'ModelParameterArguments'	Parameters in the model workspace configured as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value.
'InternalData'	Data elements that are internal to a model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `setData(myCPPMappingObj, 'Inports', 'DataVisibility', 'private')`

MemberAccessMethod — How methods are generated for data elements

'Method' | 'Inlined method' | 'Structure-based method' | 'Inlined structure-based method' | 'None'

How the methods, if any, are generated for the data elements. This configuration controls how application code can view and modify the class member data.

Member Access Method	Description
'Method'	The <code>get</code> and <code>set</code> methods for each element in the category appear in the generated class. Only <code>set</code> methods appear for model elements in the <code>Inports</code> category. Only <code>get</code> methods appear for model elements in the <code>Outports</code> category.
'Inlined method'	The <code>get</code> and <code>set</code> methods defined in their declarations appear for each element in a category in the generated class. Only <code>set</code> methods appear for model elements in the <code>Inports</code> category. Only <code>get</code> methods appear for model elements in the <code>Outports</code> category.

Member Access Method	Description
'Structure-based method'	Data elements appear as a structure in the class and aggregate <code>get</code> and <code>set</code> methods are generated for each category.
'Inlined structure-based method'	Data elements appear as a structure in the class. Aggregate <code>get</code> and <code>set</code> methods defined in their declaration are generated for each category.
'None'	If you configure the access of a model element category to <code>None</code> , <code>get</code> and <code>set</code> methods do not appear in the generated class. The application code can directly access the data.

Data Types: `char` | `string`

DataVisibility — Visibility of data category in generated code

'private' | 'public' | 'protected'

The visibility of the data category in generated code, specified as either `private`, `public`, or `protected`. If you configure data elements as `public`, they appear as public members of the generated class. If you configure elements as `private`, they appear as private members of the generated class.

Data Types: `char` | `string`

DataAccess — Access of inports, outports, or model parameter arguments in generated code

'Direct' (default) | 'Pointer'

Data access of inports, outports, or model parameter arguments in the generated code, specified as either 'Pointer' or 'Direct'.

You can configure data access for 'Inports', 'Outports', and 'ModelParameterArguments' categories:

- 'Inports' whose member access method is set to 'Structure-based method' or 'Inlined structure-based method'.
- 'Outports' whose member access method is set to 'Structure-based method' or 'Inlined structure-based method'.
- 'ModelParameterArguments' whose data visibility is set to 'private' (not 'Individual Arguments').

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`find` | `getClassName` | `setClassName` | `getClassNamespace` | `setClassNamespace` | `getData` | `getFunction` | `setFunction`

Topics

"Interactively Configure C++ Interface"

"Programmatically Configure C++ Interface"

getFunction

Get code configuration from code mappings for model function

Syntax

```
propertyValue = getFunction(myCPMappingObj,methodID,property)
```

Description

`propertyValue = getFunction(myCPMappingObj,methodID,property)` returns the value of a property for the specified model function.

Examples

Get the Method Name for the Initialize Function for a Model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('CppClassRateBased');  
cm = coder.mapping.api.get('CppClassRateBased');
```

To access the method name of the initialize function for the model, use the `getFunction` function. If you did not specify a method name for the initialize function, `getFunction` returns an empty character vector, and the method name in the generated code uses the default method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')  
  
value =  
'initIntegrator'
```

Specify a method name for the initialize function by using the `setFunction` function.

```
setFunction(cm, 'Initialize', 'MethodName', 'my_$N')
```

The `getFunction` function now returns the specified method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')  
  
value =  
'my_$N'
```

Input Arguments

myCPMappingObj — C++ code mapping object

`CodeMappingCPP` object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

methodID — Model method identifier

string scalar | character vector

The identifier of the model method to get the code mappings property value of, specified as one of the values in the table.

Type of Model Method	Method Identifier
Initialize method	"Initialize"
Terminate method	"Terminate"
Reset method	"Reset: <i>sIdentifier</i> ", where <i>sIdentifier</i> is the name of the reset method in the model
Periodic method	"Periodic:<i>sIdentifier</i>", where <i>sIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "Periodic:D1" Tip For a single-tasking periodic method, you can use "Periodic" without <i>:sIdentifier</i>.
Periodic update method	"PeriodicUpdate:<i>sIdentifier</i>", where <i>sIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "PeriodicUpdate:D1". Tip For a single-tasking periodic update method, you can use "PeriodicUpdate" without <i>:sIdentifier</i>.
Partition method	"Partition:<i>sIdentifier</i>", where <i>sIdentifier</i> is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "Partition:P1"
Partition update method	"PartitionUpdate:<i>sIdentifier</i>", where <i>sIdentifier</i> is a partition created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "PartitionUpdate:P1"
Simulink function	"SimulinkFunction: <i>sIdentifier</i> ", where <i>sIdentifier</i> is the name of the Simulink function in the model
Exported method	"ExportedFunction: <i>sIdentifier</i> ", where <i>sIdentifier</i> is the name of the Inport block that drives the control port of the function-call subsystem, or the name of the <i>port</i> if it is driven by an In Bus Element block

Tip When the configuration parameter **Single output/update function** is cleared, you can specify the update method that corresponds to a partition, periodic multitasking, or periodic single-tasking

method. For more information about generating code with a single method for both update and output, see **Single output/update function**.

For information about model partitioning, see “Create Partitions”. To learn more about annotations and sample time information, see “Specify Sample Time”. To learn how to programmatically obtain sample time information, see `simulink.schedule.OrderedSchedule`.

property — Code mapping property value to return

MethodName | Arguments

Code mapping property value to return. Specify one of the property names listed in this table.

Information to Return	Property Name
Name to use for the method in the generated code	MethodName
For periodic, single-tasking functions and Simulink functions, a string that shows the block names, argument names, type qualifiers, and order of arguments as they will appear in the generated code	Arguments

Example: 'MethodName'

Output Arguments

propertyValue — Name of function, or argument specification

character vector | string scalar

Name of the function or argument specification returned as a character vector or string scalar.

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`find` | `getClassName` | `setClassName` | `getClassNamespace` | `setClassNamespace` | `getData` | `setData` | `setFunction`

Topics

“Interactively Configure C++ Interface”

“Programmatically Configure C++ Interface”

setFunction

Set code mapping information for model function

Syntax

```
setFunction(myCPPMappingObj, funcID, Name, Value)
```

Description

`setFunction(myCPPMappingObj, funcID, Name, Value)` sets code mapping information for the specified model function. Use this function to set the method name for a model function. For single-tasking periodic functions and Simulink functions, you can use this function to set the argument specification, including argument names, type qualifiers, and argument order.

Examples

Set the Method Name for the Initialize Function for a Model

Open the model. To access the `CodeMappingCPP` object associated with the model, use the `coder.mapping.api.get` function.

```
open_system('CppClassRateBased');
cm = coder.mapping.api.get('CppClassRateBased');
```

To access the method name of the initialize function for the model, use the `getFunction` function. If you did not specify a method name for the initialize function, the `getFunction` function returns an empty character vector, and the method name in the generated code uses the default method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')

value =
'initIntegrator'
```

Specify a method name for the initialize function by using the `setFunction` function.

```
setFunction(cm, 'Initialize', 'MethodName', 'my_$N')
```

The `getFunction` function now returns the specified method name.

```
value = getFunction(cm, 'Initialize', 'MethodName')

value =
'my_$N'
```

Input Arguments

myCPPMappingObj — C++ code mapping object
CodeMappingCPP object

C++ code mapping object, returned by a call to either the `coder.mapping.utils.create` function or the `coder.mapping.api.get` function.

methodID — Model method identifier

string scalar | character vector

The identifier of the model method to set the code mappings property value for, specified as one of the values in the table.

Type of Model Method	Method Identifier
Initialize method	"Initialize"
Terminate method	"Terminate"
Reset method	"Reset: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the reset method in the model
Periodic method	"Periodic:<i>slIdentifier</i>", where <i>slIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "Periodic:D1" Tip For a single-tasking periodic method, you can use "Periodic" without <i>:slIdentifier</i>.
Periodic update method	"PeriodicUpdate:<i>slIdentifier</i>", where <i>slIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "PeriodicUpdate:D1". Tip For a single-tasking periodic update method, you can use "PeriodicUpdate" without <i>:slIdentifier</i>.
Partition method	"Partition:<i>slIdentifier</i>", where <i>slIdentifier</i> is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "Partition:P1"
Partition update method	"PartitionUpdate:<i>slIdentifier</i>", where <i>slIdentifier</i> is a partition created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "PartitionUpdate:P1"
Simulink function	"SimulinkFunction: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the Simulink function in the model
Exported method	"ExportedFunction: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the Inport block that drives the control port of the function-call subsystem, or the name of the <i>port</i> if it is driven by an In Bus Element block

Tip When the configuration parameter **Single output/update function** is cleared, you can specify the update method that corresponds to a partition, periodic multitasking, or periodic single-tasking method. For more information about generating code with a single method for both update and output, see **Single output/update function**.

For information about model partitioning, see “Create Partitions”. To learn more about annotations and sample time information, see “Specify Sample Time”. To learn how to programmatically obtain sample time information, see `simulink.schedule.OrderedSchedule`.

Name-Value Pair Arguments

Specify 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 as `Name1, Value1, ..., NameN, ValueN`. The order of the name and value pair arguments does not matter.

Example: `'MethodName', 'my_$N'`

MethodName — Name of method

character vector | string scalar

Name for the entry-point method in the generated C++ code, specified as a character vector or string scalar.

Data Types: `char` | `string`

Arguments — Argument specification

character vector | string scalar

Argument specification for the entry-point method in the generated C++ code, specified as a character vector or string scalar. The specification is a method prototype that shows argument names, type qualifiers, and argument order, for example, `'y =(u1, const *u2)'`.

For an entry-point method that does not return a value, do not include an output argument in the prototype specification. For example,

```
setFunction(cm, 'SimulinkFunction:f', 'Arguments', '(u, *y)');
```

When the model function specified is a periodic function, you can also customize parameter names in the argument specification.

```
setFunction(cm, 'Periodic:D1', 'Arguments', ...
'(In1_1s & myParam1, In2_2s arg_In2_2s, * Out1 arg_Out1)');
```

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`find` | `getClassName` | `setClassName` | `getClassNamespaces` | `setClassNamespaces` | `getData` | `setData` | `getFunction`

Topics

"Interactively Configure C++ Interface"

"Programmatically Configure C++ Interface"

setFunctionDefault

Set default function customization template and memory section for model functions category

Syntax

```
setFunctionDefault(myCoderDictionaryObj,category,Name,Value)
```

Description

`setFunctionDefault(myCoderDictionaryObj,category,Name,Value)` sets the default function customization template and memory section for the specified category of model entry-point functions.

Examples

Configure the default memory section for a data category

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get("codeDefinitions.sldd");
```

To see the memory section for Execution functions in the dictionary, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm,"Execution","MemorySection")
```

```
value =
```

```
    'None'
```

To configure the memory section for the category, use the `setFunctionDefault` function.

```
setFunctionDefault(cm,"Execution",MemorySection="functionFastMem")
```

To verify that the memory section for the Execution category is now set to `functionFastMem`, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm,"Execution","MemorySection")
```

```
value =
```

```
    'functionFastMem'
```

Input Arguments

myCoderDictionaryObj — Coder dictionary object

`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

category — Model function category`InitializeTerminate | Execution | SharedUtility`

Category of model entry-point functions for which to set the function customization template and memory section.

Example: "Execution"

Name-Value Pair Arguments

Specify 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 as `Name1, Value1, ..., NameN, ValueN`. The order of the name and value pair arguments does not matter.

Example: `FunctionCustomizationTemplate="exFastFunction"`

FunctionCustomizationTemplate — Name of function customization template`character vector | string scalar`

Name of a function customization template defined in the Embedded Coder Dictionary associated with the model. If you set the default function customization template for a category of functions to `Default`, you can specify a memory section for the category of functions.

Data Types: `char` | `string`

MemorySection — Name of memory section`None | MemConst | MemVolatile | MemConstVolatile | internalDataMem | functionFastMem | functionSlowMem`

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`coder.mapping.api.CoderDictionary | setDataDefault | getDataDefault | getFunctionDefault`

getFunctionDefault

Get default function customization template or memory section for model functions category

Syntax

```
propertyValue = getFunctionDefault(myCoderDictionaryObj,category,property)
```

Description

`propertyValue = getFunctionDefault(myCoderDictionaryObj,category,property)`
returns the value of the specified property for the specified function category.

Examples

Configure the default memory section for a data category

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get("codeDefinitions.sldd");
```

To see the memory section for Execution functions in the dictionary, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm,"Execution","MemorySection")
```

```
value =
```

```
    'None'
```

To configure the memory section for the category, use the `setFunctionDefault` function.

```
setFunctionDefault(cm,"Execution",MemorySection="functionFastMem")
```

To verify that the memory section for the Execution category is now set to `functionFastMem`, use the `getFunctionDefault` function.

```
value = getFunctionDefault(cm,"Execution","MemorySection")
```

```
value =
```

```
    'functionFastMem'
```

Input Arguments

myCoderDictionaryObj — Coder dictionary object

`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

category — Model function category

`InitializeTerminate` | `Execution` | `SharedUtility`

Category of model entry-point functions for which to set the function customization template and memory section.

Example: "Execution"

property — Function customization template or memory section

`FunctionCustomizationTemplate` | `MemorySection`

`FunctionCustomizationTemplate` or `MemorySection` for which to return a value.

Example: "FunctionCustomizationTemplate"

Output Arguments

propertyValue — Name of function customization template or memory section

character vector | string scalar

Name of the function customization template or memory section.

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`coder.mapping.api.CoderDictionary` | `setDataDefault` | `getDataDefault` | `setFunctionDefault`

getFunction

Get code configuration from code mappings for model function

Syntax

```
propertyValue = getFunction(coderMapObj,funcID,funcProperty)
```

Description

`propertyValue = getFunction(coderMapObj,funcID,funcProperty)` returns the value of a property for the specified model function. Use this function to return the function customization template or memory section configured for a model function. For single-tasking periodic functions for which you previously set an argument specification and for Simulink functions, use this function to return the argument specification.

Examples

Programmatically Get and Set Function Properties in Code Mappings of Simulink® Models

Use the programmatic interface to retrieve and configure properties of functions in the code mappings of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure the **Code Mappings** editor is open with the **Functions** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model `ECoderMapAPI`.

```
simulinkModel = "ECoderMapAPI";
open_system(simulinkModel);
```

Get the code mappings object of the model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Get the `FunctionName` property of the periodic function and the initialization function.

```
periodicFcnName = getFunction(codeMapObj,"Periodic:D1","FunctionName")
```

```
periodicFcnName =
'myPeriodicFcn'
```

```
InitFcnName = getFunction(codeMapObj,"Initialize","FunctionName")
```

```
InitFcnName =
'myInitFcn'
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Entry-point functions are declared in the model header file. Store the header file name in the variable `model_h_file`.

```
model_h_file = fullfile(simulinkModel+"_ert_rtw",simulinkModel+".h")

model_h_file =
"ECoderMapAPI_ert_rtw\ECoderMapAPI.h"
```

This is the declaration of the entry-point functions in the header file:

```
/* Model entry point functions */
extern void myInitFcn(void);
extern void myPeriodicFcn(void);
```

The function names are the names stored in `periodicFcnName` and `InitFcnName`.

To open the header file, enter this command in the MATLAB® Command Window:

```
edit(model_h_file)
```

Rename the periodic functions.

```
if(startsWith(periodicFcnName,"my"))
    setFunction(codeMapObj,"Periodic:D1",FunctionName="yourPeriodicFcn");
else
    setFunction(codeMapObj,"Periodic:D1",FunctionName="myPeriodicFcn");
end
if(startsWith(InitFcnName,"my"))
    setFunction(codeMapObj,"Initialize",FunctionName="yourInitFcn");
else
    setFunction(codeMapObj,"Initialize",FunctionName="myInitFcn");
end
```

Generate code from the model again with the new entry-point function names.

```
evalc("slbuild(simulinkModel)");
```

This is the updated declaration of the entry-point functions in the header file.

```
/* Model entry point functions */
extern void yourInitFcn(void);
extern void yourPeriodicFcn(void);
```

The function names are updated.

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

funcID — Model function identifier

string scalar | character vector

The identifier of the model function to get the code mappings property value of, specified as one of the values in the table.

Type of Model Function	Function Identifier
Initialize function	"Initialize"
Terminate function	"Terminate"
Reset function	"Reset: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the reset function in the model
Periodic function	"Periodic:<i>slIdentifier</i>", where <i>slIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "Periodic:D1" Tip For a single-tasking periodic function, you can use "Periodic" without <i>:slIdentifier</i>.
Periodic update function	"PeriodicUpdate:<i>slIdentifier</i>", where <i>slIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "PeriodicUpdate:D1". Tip For a single-tasking periodic update function, you can use "PeriodicUpdate" without <i>:slIdentifier</i>.
Partition function	"Partition:<i>slIdentifier</i>", where <i>slIdentifier</i> is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "Partition:P1"
Partition update function	"PartitionUpdate:<i>slIdentifier</i>", where <i>slIdentifier</i> is a partition created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "PartitionUpdate:P1"
Simulink function	"SimulinkFunction: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the Simulink function in the model
Exported function	"ExportedFunction: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the Inport block that drives the control port of the function-call subsystem, or the name of the <i>port</i> if it is driven by an In Bus Element block

Tip When the configuration parameter **Single output/update function** is cleared, you can specify the update function that corresponds to a partition, periodic multitasking, or periodic single-tasking function. For more information about generating code with a single function for both update and output, see **Single output/update function**.

For information about model partitioning, see “Create Partitions”. To learn more about annotations and sample time information, see “Specify Sample Time”. To learn how to programmatically obtain sample time information, see `simulink.schedule.OrderedSchedule`.

funcProperty — Code mapping property value to return

`FunctionCustomizationTemplate` | `MemorySection` | `FunctionName` | `Arguments` | `TimerService`

Code mapping property value to return. Specify one of the property names listed in this table.

Information to Return	Property Name
Function customization template setting for the specified function	<code>FunctionCustomizationTemplate</code>
Memory section associated with the specified function	<code>MemorySection</code>
Name to use for the function in the generated code	<code>FunctionName</code>
For periodic, single-tasking functions and Simulink functions, a string that shows the names, type qualifiers, and order of arguments as they will appear in the generated code	<code>Arguments</code>
Name of timer service defined in Embedded Coder Dictionary	<code>TimerService</code>

Example: `"FunctionCustomizationTemplate"`

Output Arguments

propertyValue — Name of function customization template, memory section, function, or argument specification

character vector | string scalar

Name of the function customization template, memory section, function, or argument specification returned as a character vector or string scalar.

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `setFunction` | `getFunctionDefault` | `setFunctionDefault`

Topics

“Configure Names for Individual C Entry-Point Functions”

“Configure Name and Arguments for Individual Step Functions”

“Configure Default C Code Generation for Categories of Data Elements and Functions”

“C Data Code Interface Configuration for Model Interface Elements”

“Configure Timer Service Interfaces”

“Create and Analyze Random Schedules for a Model Using the Schedule Editor API”

getFunctionDefault

Get default function customization template or memory section for model functions category

Syntax

```
propertyValue = getFunctionDefault(coderMapObj, funcCategory, funcProperty)
```

Description

`propertyValue = getFunctionDefault(coderMapObj, funcCategory, funcProperty)` returns the value of the specified property for the specified function category.

You cannot specify default function interfaces for models with an attached Embedded Coder Dictionary that defines a service interface configuration.

Examples

Programmatically Get and Set Function Defaults in Code Mappings of Simulink® Models

Use the programmatic interface to retrieve and configure function defaults in the code mappings of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor and the model dictionary, make sure the **Code Mappings** editor is open with the **Function Defaults** tab selected, and the model dictionary open with the **Function Customization Template** section selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 . To learn how to open the model dictionary, see “Open the Embedded Coder Dictionary” on page 23-0 .

Open the model `CoderMapAPI`.

```
simulinkModel = "ECoderMapAPI";  
open_system(simulinkModel);
```

Get the code mappings object of this model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Specify `FunctionName` as an empty string for all model functions so that their names in the generated code are determined by the value you set for the function defaults.

```
setFunction(codeMapObj, find(codeMapObj, "Functions"), FunctionName=string.empty);
```

Get the default customization template of the `InitializeTerminate` and `Execution` functions.

```
exeFCNsTemplate = getFunctionDefault(codeMapObj, "Execution", "FunctionCustomizationTemplate")
```

```
exeFCNsTemplate =  
'FCN_Template__2'
```

```
initTermFCNsTemplate = getFunctionDefault(codeMapObj, "InitializeTerminate", "FunctionCustomization
```



```
initTermFCNsTemplate =
'FCN_Template__1'
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Model entry-point functions are declared in the generated header file of the model. Store the header file name.

```
model_h_file = fullfile(simulinkModel+"_ert_rtw",simulinkModel+".h")
```

```
model_h_file =
"ECoderMapAPI_ert_rtw\ECoderMapAPI.h"
```

Function Naming Rule of FCN_Template__1 begins with Template_1 and **Function Naming Rule** of FCN_Template__2 begins with Template_2. This is the entry-point function declaration section in the generated header file:

```
/* Model entry point functions */
extern void Template_1_FCN_ECoderMapAPI_initialize(void);
extern void Template_2_FCN_ECoderMapAPI_step(void);
extern void Template_1_FCN_ECoderMapAPI_terminate(void);
```

The function names are generated according to the specified **Function Customization Template** values for those functions.

To open the header file and, enter this command in the MATLAB® Command Window:

```
edit(model_h_file)
```

Switch between the templates of the two function categories.

```
setFunctionDefault(codeMapObj,"Execution",FunctionCustomizationTemplate=initTermFCNsTemplate);
setFunctionDefault(codeMapObj,"InitializeTerminate",FunctionCustomizationTemplate=exeFCNsTemplate);
```

Generate code from the model again.

```
evalc("slbuild(simulinkModel)");
```

The entry-point function names are updated according to their updated **Function Customization Templates**:

```
/* Model entry point functions */
extern void Template_2_FCN_ECoderMapAPI_initialize(void);
extern void Template_1_FCN_ECoderMapAPI_step(void);
extern void Template_2_FCN_ECoderMapAPI_terminate(void);
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

funcCategory — Model function category

"InitializeTerminate" | "Execution" | "SharedUtility"

Category of model entry-point functions for which to return the default function customization template or memory section.

Example: "Execution"

funcProperty — Function customization template or memory section

"FunctionCustomizationTemplate" | "MemorySection"

FunctionCustomizationTemplate or MemorySection for which to return a value.

Example: "FunctionCustomizationTemplate"

Output Arguments

propertyValue — Name of function customization template or memory section

character vector | string scalar

Name of the function customization template or memory section.

Data Types: char | string

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getFunction` | `setFunction` | `setFunctionDefault`

Topics

"Configure Default C Code Generation for Categories of Data Elements and Functions"

"C Data Code Interface Configuration for Model Interface Elements"

setFunction

Set coder mapping information for model function

Syntax

```
setFunction(coderMapObj, funcID, Name=Value)
```

Description

`setFunction(coderMapObj, funcID, Name=Value)` sets code mapping information for the specified model function. Use this function to set the function customization template, memory section, or function name for a model function. For single-tasking periodic functions and Simulink functions, you can use this function to set the argument specification, including argument names, type qualifiers, and argument order.

Examples

Programmatically Get and Set Function Properties in Code Mappings of Simulink® Models

Use the programmatic interface to retrieve and configure properties of functions in the code mappings of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure the **Code Mappings** editor is open with the **Functions** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model `ECoderMapAPI`.

```
simulinkModel = "ECoderMapAPI";
open_system(simulinkModel);
```

Get the code mappings object of the model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Get the `FunctionName` property of the periodic function and the initialization function.

```
periodicFcnName = getFunction(codeMapObj, "Periodic:D1", "FunctionName")
```

```
periodicFcnName =
'myPeriodicFcn'
```

```
InitFcnName = getFunction(codeMapObj, "Initialize", "FunctionName")
```

```
InitFcnName =
'myInitFcn'
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Entry-point functions are declared in the model header file. Store the header file name in the variable `model_h_file`.

```
model_h_file = fullfile(simulinkModel+"_ert_rtw",simulinkModel+".h")

model_h_file =
"ECoderMapAPI_ert_rtw\ECoderMapAPI.h"
```

This is the declaration of the entry-point functions in the header file:

```
/* Model entry point functions */
extern void myInitFcn(void);
extern void myPeriodicFcn(void);
```

The function names are the names stored in `periodicFcnName` and `InitFcnName`.

To open the header file, enter this command in the MATLAB® Command Window:

```
edit(model_h_file)
```

Rename the periodic functions.

```
if(startsWith(periodicFcnName,"my"))
    setFunction(codeMapObj,"Periodic:D1",FunctionName="yourPeriodicFcn");
else
    setFunction(codeMapObj,"Periodic:D1",FunctionName="myPeriodicFcn");
end
if(startsWith(InitFcnName,"my"))
    setFunction(codeMapObj,"Initialize",FunctionName="yourInitFcn");
else
    setFunction(codeMapObj,"Initialize",FunctionName="myInitFcn");
end
```

Generate code from the model again with the new entry-point function names.

```
evalc("slbuild(simulinkModel)");
```

This is the updated declaration of the entry-point functions in the header file.

```
/* Model entry point functions */
extern void yourInitFcn(void);
extern void yourPeriodicFcn(void);
```

The function names are updated.

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object of the model whose function is to be set, specified as a `coder.mapping.api.CodeMapping` object.

Example: `myCM`

funcID — Model function identifier

string scalar | character vector

The identifier of the model function to set the code mappings property value for, specified as one of the values in the table.

Type of Model Function	Function Identifier
Initialize function	"Initialize"
Terminate function	"Terminate"
Reset function	"Reset: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the reset function in the model
Periodic function	"Periodic:<i>slIdentifier</i>", where <i>slIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "Periodic:D1" Tip For a single-tasking periodic function, you can use "Periodic" without <i>:slIdentifier</i>.
Periodic update function	"PeriodicUpdate:<i>slIdentifier</i>", where <i>slIdentifier</i> is an annotation that corresponds to the sample time period for a periodic or continuous rate Example: "PeriodicUpdate:D1". Tip For a single-tasking periodic update function, you can use "PeriodicUpdate" without <i>:slIdentifier</i>.
Partition function	"Partition:<i>slIdentifier</i>", where <i>slIdentifier</i> is a partition that was created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "Partition:P1"
Partition update function	"PartitionUpdate:<i>slIdentifier</i>", where <i>slIdentifier</i> is a partition created explicitly from a block in the model and shown in the Simulink Schedule Editor Example: "PartitionUpdate:P1"
Simulink function	"SimulinkFunction: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the Simulink function in the model
Exported function	"ExportedFunction: <i>slIdentifier</i> ", where <i>slIdentifier</i> is the name of the Inport block that drives the control port of the function-call subsystem, or the name of the <i>port</i> if it is driven by an In Bus Element block

Tip When the configuration parameter **Single output/update function** is cleared, you can specify the update function that corresponds to a partition, periodic multitasking, or periodic single-tasking function. For more information about generating code with a single function for both update and output, see **Single output/update function**.

For information about model partitioning, see “Create Partitions”. To learn more about annotations and sample time information, see “Specify Sample Time”. To learn how to programmatically obtain sample time information, see `simulink.schedule.OrderedSchedule`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `FunctionCustomizationTemplate="exFastFunction"`

FunctionCustomizationTemplate — Name of function customization template

character vector | string scalar

Name of a function customization template defined in the Embedded Coder Dictionary associated with the model, specified as a character vector or string scalar. If you set the default function customization template for a category of functions to `Default`, you can specify a memory section for the functions category.

Data Types: `char` | `string`

MemorySection — Name of memory section

character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model, specified as a character vector or string scalar.

Data Types: `char` | `string`

FunctionName — Name of function

character vector | string scalar

Name for the entry-point function in the generated C code, specified as a character vector or string scalar.

Data Types: `char` | `string`

Arguments — Argument specification

character vector | string scalar

Argument specification for the entry-point function in the generated C code, specified as a character vector or string scalar. The specification is a function prototype that shows argument names, type qualifiers, and argument order (for example, `y=(u1, const *u2)`).

Data Types: `char` | `string`

TimerService — Name of timer service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a timer service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify `"Dictionary default"`.

This property is only applicable for exported functions, and only for service interface mappings. For more information, see “Configure Timer Service Interfaces”.

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getFunction` |
`getFunctionDefault` | `setFunctionDefault`

Topics

"Programmatically Customize C Functions Generated from Global Simulink Functions"
"Programmatically Customize C Functions Generated from Scoped Simulink Functions Located in Top Model"
"Programmatically Customize Generated Entry-Point C Function Names"
"Programmatically Customize Arguments of Step C Function Generated from Single Tasking Model"
"Programmatically Customize Generated Entry-Point C Function Arguments for Component Models"
"Configure Names for Individual C Entry-Point Functions"
"Configure Name and Arguments for Individual Step Functions"
"Configure Default C Code Generation for Categories of Data Elements and Functions"
"C Data Code Interface Configuration for Model Interface Elements"
"Configure Timer Service Interfaces"
"Create and Analyze Random Schedules for a Model Using the Schedule Editor API"

setFunctionDefault

Set default function customization template and memory section for model functions category

Syntax

```
setFunctionDefault(coderMapObj, funcCategory, Name=Value)
```

Description

`setFunctionDefault(coderMapObj, funcCategory, Name=Value)` sets the default function customization template and memory section for the specified category of model entry-point functions.

You cannot specify default function interfaces for models with an attached Embedded Coder Dictionary that defines a service interface configuration.

Examples

Programmatically Get and Set Function Defaults in Code Mappings of Simulink® Models

Use the programmatic interface to retrieve and configure function defaults in the code mappings of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor and the model dictionary, make sure the **Code Mappings** editor is open with the **Function Defaults** tab selected, and the model dictionary open with the **Function Customization Template** section selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 . To learn how to open the model dictionary, see “Open the Embedded Coder Dictionary” on page 23-0 .

Open the model `CoderMapAPI`.

```
simulinkModel = "ECoderMapAPI";  
open_system(simulinkModel);
```

Get the code mappings object of this model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Specify `FunctionName` as an empty string for all model functions so that their names in the generated code are determined by the value you set for the function defaults.

```
setFunction(codeMapObj, find(codeMapObj, "Functions"), FunctionName=string.empty);
```

Get the default customization template of the `InitializeTerminate` and `Execution` functions.

```
exeFCNsTemplate = getFunctionDefault(codeMapObj, "Execution", "FunctionCustomizationTemplate")
```

```
exeFCNsTemplate =  
'FCN_Template__2'
```

```
initTermFCNsTemplate = getFunctionDefault(codeMapObj, "InitializeTerminate", "FunctionCustomization
```



```
initTermFCNsTemplate =
'FCN_Template__1'
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Model entry-point functions are declared in the generated header file of the model. Store the header file name.

```
model_h_file = fullfile(simulinkModel+"_ert_rtw",simulinkModel+".h")
```

```
model_h_file =
"ECoderMapAPI_ert_rtw\ECoderMapAPI.h"
```

Function Naming Rule of FCN_Template__1 begins with Template_1 and **Function Naming Rule** of FCN_Template__2 begins with Template_2. This is the entry-point function declaration section in the generated header file:

```
/* Model entry point functions */
extern void Template_1_FCN_ECoderMapAPI_initialize(void);
extern void Template_2_FCN_ECoderMapAPI_step(void);
extern void Template_1_FCN_ECoderMapAPI_terminate(void);
```

The function names are generated according to the specified **Function Customization Template** values for those functions.

To open the header file and, enter this command in the MATLAB® Command Window:

```
edit(model_h_file)
```

Switch between the templates of the two function categories.

```
setFunctionDefault(codeMapObj,"Execution",FunctionCustomizationTemplate=initTermFCNsTemplate);
setFunctionDefault(codeMapObj,"InitializeTerminate",FunctionCustomizationTemplate=exeFCNsTemplate);
```

Generate code from the model again.

```
evalc("slbuild(simulinkModel)");
```

The entry-point function names are updated according to their updated **Function Customization Templates**:

```
/* Model entry point functions */
extern void Template_2_FCN_ECoderMapAPI_initialize(void);
extern void Template_1_FCN_ECoderMapAPI_step(void);
extern void Template_2_FCN_ECoderMapAPI_terminate(void);
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

funcCategory — Model function category

InitializeTerminate | Execution | SharedUtility

Category of model entry-point functions for which to set the function customization template and memory section.

Example: "Execution"

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `FunctionCustomizationTemplate="exFastFunction"`

FunctionCustomizationTemplate — Name of function customization template

character vector | string scalar

Name of a function customization template defined in the Embedded Coder Dictionary associated with the model. If you set the default function customization template for a category of functions to `Default`, you can specify a memory section for the category of functions.

Data Types: `char` | `string`

MemorySection — Name of memory section

character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getFunction` | `getFunctionDefault` | `setFunction`

Topics

"Programmatically Customize C Functions Generated from Scoped Simulink Functions Located in Top Model"

"Configure Default C Code Generation for Categories of Data Elements and Functions"

"C Data Code Interface Configuration for Model Interface Elements"

instrumentCode

Add instrumentation to generated code to perform execution time / memory usage profiling and analyze code coverage

Syntax

```
instrumentCode(buildFolder)
instrumentCode( ___,Name=Value)
```

Description

`instrumentCode(buildFolder)` generates a SIL or PIL MEX function for the code generation files located in `buildFolder` that you already produced by using the `codegen` command. The generated MEX does not contain any additional instrumentation.

`instrumentCode( ___,Name=Value)` enables you to include additional instrumentation in the SIL or PIL MEX that you generate using the previous syntax. Provide one or more name-value arguments to specify the instrumentation that you intend to include in the generated MEX. You can add instrumentation to track stack memory usage, analyze C/C++ code coverage, profile entry-point functions, or profile functions called within entry-point functions. You can also use `instrumentCode` to specify the toolchain to use to build the generated code, and compiler optimization and debug settings for the specified toolchain.

The `instrumentCode` function decouples the code generation and code instrumentation steps. This functionality enables you to analyze the execution behavior of the code you intend to deploy without altering the original functional code. Once you generate the code, you can apply the instrumentation as many times as you need.

Examples

Add Instrumentation That Performs Coverage Analysis

This example shows how to use the `instrumentCode` function to add instrumentation for C/C++ coverage analysis to code that you already generated by using the `codegen` command. To perform coverage analysis, you need a MATLAB® Test™ license.

Define Entry-Point Function and Generate Code

Define a MATLAB entry-point function `coverageAnalysisExample` that performs a binary operation on its first two inputs. The binary operation depends on whether the third input is positive, negative, or zero.

```
type coverageAnalysisExample

function out = coverageAnalysisExample(A,B,flag) %#codegen
if flag > 0
    out = A + B;
elseif flag < 0
    out = A - B;
```

```
else
    out = A * B;
end
end
```

Generate static library code for software-in-the-loop (SIL) execution. Specify all inputs to be of scalar double type. Specify the name of the code generation folder to be `my_codegen_folder`.

```
cfg = coder.config('lib');
cfg.VerificationMode = 'SIL';
codegen -config cfg coverageAnalysisExample -args {0,0,0} -d my_codegen_folder
```

Code generation successful.

Add Instrumentation to Generated Code

Add C/C++ code coverage instrumentation to the generated code and produce an instrumented MEX.

```
instrumentCode('my_codegen_folder', 'CodeCoverage', true)
```

Completed instrumentation of generated code.

Execute the MEX and View Coverage Data

Execute `coverageAnalysisExample_sil` with two sets of sample inputs. Make sure the third input is positive for one set and negative for the other.

```
coverageAnalysisExample_sil(10,4,3)

### Starting SIL execution for 'coverageAnalysisExample'
To terminate execution: clear coverageAnalysisExample_sil

ans =
14

coverageAnalysisExample_sil(10,4,-2)

ans =
6
```

Terminate SIL execution and view the code coverage report.

```
clear coverageAnalysisExample_sil

### Application stopped
### Stopping SIL execution for 'coverageAnalysisExample'
Code coverage report: getCodeCoverageData('coverageAnalysisExample_sil')

getCodeCoverageData('coverageAnalysisExample_sil');
```

Code Coverage Report

The code coverage report provides a detailed analysis of the source code covered by the tests.

Overall Coverage Summary

Summary of the code coverage metrics for all source files.

Total Files				
3	Coverage...	Executable	Missed	Code Co...
	Function	3	0	100%
	Statement	7	1	85.71%

Currently viewing: Statement

Breakdown by Source

Code coverage metrics per source file.

Summary View Detailed View

Root Folder - C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc23a.j2172582\ecoder-ex59520466\my_codegen_folder\

	File Name	Function	Statement
1	coverageAnalysisExample.c	100%	85.71%
2	coverageAnalysisExample_initialize.c	100%	N/A
3	coverageAnalysisExample_terminate.c	100%	N/A

Because neither of your test runs had the third input argument equal to zero, the last branch of the `if-else` block was never executed in the generated C entry-point file `coverageAnalysisExample.c`. This caused the `coverageAnalysisExample.c` file to have less than 100% statement coverage.

Input Arguments

buildFolder — Location of code generation files

character vector | string scalar

Absolute or relative path to the folder that contains the code generation files. This is the folder path that you specify with the `-d` flag when using the `codegen` command.

Example: `"my_codegen_folder"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Note These name-value pair arguments, except `BuildMethod`, are also available as properties in the `coder.EmbeddedCodeConfig` object. The instrumented MEX generated by using the

`instrumentCode` function has the same behavior as the MEX generated with the corresponding code configuration settings and the `codegen` command.

Example: `instrumentCode("my_codegen_folder", CodeExecutionProfiling = true, CodeProfilingInstrumentation = true)`

VerificationMode — Target that executes generated code

"SIL" (default) | "PIL"

Specify whether you want to generate MEX for software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution.

Data Types: `char` | `string`

CodeStackProfiling — Stack usage profiling during a SIL or PIL execution

false (default) | true

Stack usage profiling of entry-point functions during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, specified as one of the values in this table.

Value	Description
false	This value is the default value. Disables stack usage profiling during a SIL or PIL execution.
true	Instruments generated code to calculate stack usage of functions in the SIL or PIL execution. Use the Code Profile Analyzer to view the stack usage values for entry-point and internal functions.

Data Types: `logical`

CodeCoverage — Analysis of C/C++ code coverage

false (default) | true

Coverage analysis of both generated C/C++ code and custom C/C++ during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, specified as one of the values in this table.

Value	Description
false	This value is the default value. Disables code coverage analysis during a SIL or PIL execution.
true	Instruments C/C++ code for coverage analysis in the SIL or PIL application.

To use this name-value argument, you must have a MATLAB Test license.

Data Types: `logical`

CodeExecutionProfiling — Execution time profiling during a SIL or PIL execution

false (default) | true

Execution-time profiling of entry-point functions during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, specified as one of the values in this table.

Value	Description
false	This value is the default value. Disables execution-time profiling during a SIL or PIL execution.
true	Instruments generated code to calculate execution times of entry-point functions in the SIL or PIL execution. Use the Code Profile Analyzer to view the call tree and corresponding execution times for the generated code.

Data Types: logical

CodeProfilingInstrumentation — Execution time profiling of functions called within entry-point functions during a SIL or PIL execution

false (default) | true

Execution-time profiling of functions that are called within entry-point functions during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, specified as one of the values in this table.

Value	Description
false	This value is the default value. Disables execution-time profiling of functions called within entry-point functions during a SIL or PIL execution.
true	Instruments generated code to calculate execution times of functions called within entry-point functions in the SIL or PIL execution. Use the Code Profile Analyzer to view the call tree and corresponding execution times for the generated code.

Data Types: logical

CodeProfilingSaveOptions — Collection and storage of execution-time measurements

"AllData" (default) | "SummaryOnly" | "MetricsOnly"

Options for collection and storage of execution-time measurements, specified as one of the values in this table.

Value	Description
"AllData"	This value is the default value. As execution runs, collects and immediately uploads profiling measurement and analysis data from target device to development computer. At the end of execution, saves all data in base workspace.
"SummaryOnly"	As execution runs, evaluates only execution-time metrics required for report and immediately uploads metrics from target device. At the end of execution, saves evaluated metrics in base workspace.
"MetricsOnly"	As execution runs, processes profiling data and stores certain metrics on target device. At the end of execution, uploads the stored metrics from target device.

Data Types: char | string

BuildMethod — Toolchain to build generated code

" " (default) | character vector or string scalar

Toolchain that you want to use to build the generated code. If you specify an empty string or character vector, the generated code is built using the original settings.

Data Types: char | string

BuildConfiguration — Compiler optimization or debug settings for toolchain

" " (default) | "Faster Builds" | "Faster Runs" | "Debug" | "Specify"

Compiler optimization or debug settings for toolchain, specified as one of the values in this table.

Value	Description
" "	Compiler optimization and debug settings are not set.
"Faster Builds"	Optimizes the build for shorter build times.
"Faster Runs"	Optimizes the build for faster running executables.
"Debug"	Optimizes the build for debugging.
"Specify"	Enables you to specify options for toolchain.

Data Types: char | string

Version History

Introduced in R2023a

See Also

codegen | coder.EmbeddedCodeConfig

linux.Targets

Interface for managing target computers

Description

A `linux.Targets` object represents the available target computers and provides access to methods to add targets, remove targets, rename targets and to create target object for a particular target computer.

Creation

`tgs=linux.Targets()` creates a targets object representing target computers.

Object Functions

<code>addTarget</code>	Add target computer to target computers object
<code>getAllTargets</code>	Get list of available target computers
<code>getTarget</code>	Get handle of target computer
<code>removeTarget</code>	Delete target computer
<code>renameTarget</code>	Rename target computer

Examples

Create a Targets Object

Create a targets object `tgs` representing the target computer.

Use the following command to create the targets object

```
tgs = linux.Targets();
```

Version History

Introduced in R2022b

linux.createApplicationPackage

Create application package for model

Syntax

```
linux.createApplicationPackage(modelName)  
linux.createApplicationPackage(modelName,CreateExeForTarget = targetName)
```

Description

`linux.createApplicationPackage(modelName)` creates an application package file with `.mldatax` extension for the specified Simulink model with source code.

`linux.createApplicationPackage(modelName,CreateExeForTarget = targetName)` creates an application package file compatible for the `targetName` target without source code and with executable.

Examples

Create Application Package

For a Simulink model with name `myModel`, create application package using the following command:

```
linux.createApplicationPackage('myModel');
```

Application package gets created with source code in the folder same as the model.

```
linux.createApplicationPackage('myModel', CreateExeForTarget='LinuxTarget1');
```

Application package gets created with executable and without source code in the folder same as the model.

Input Arguments

modelName — Name of the Simulink model

character vector | string scalar

Specify the model name to create the application package with source code.

Example: `myModel`

CreateExeForTarget — Create executable for target

character vector | string scalar

Specify a target name for the `CreateExeForTarget` argument to create application with executable and without source code.

Example: `CreateExeForTarget = LinuxTarget1`

Version History

Introduced in R2022b

See Also

`deployApplicationPackage` | `removeApplicationPackage` | Linux Runtime Manager

Topics

"Build Simulink Model and Deploy Application"

addTarget

Add target computer to target computers object

Syntax

```
addTarget(tgs, targetName, ipAddress, userName, password)
```

Description

`addTarget(tgs, targetName, ipAddress, userName, password)` adds the specified target to the object.

Examples

Add a target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1`, password `myPassword`, and IP address.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Input Arguments

tgs — Targets object

handle

Target object created by the `linux.Targets` class.

Example: `tgs, target_object`

targetName — Target name

character vector | string scalar

Specify a name for the target.

Example: `"LinuxTarget1"`

ipAddress — IP address of the target

character vector | string scalar

Specify the IP address of the target.

Example: `"178.10.10.1"`

userName — User name of the target

character vector | string scalar

Specify the user name for the target.

Example: `"user1"`

password — Password for the respective user

character vector | string scalar

Specify the password for the user on the target.

Example: "myPassword"

Version History

Introduced in R2022b

See Also

linux.Targets | Linux Runtime Manager | removeTarget

Topics

"Set Up Linux Target Computer"

connect

Connect Simulink to selected target computer

Syntax

```
connect(tg)
```

Description

`connect(tg)` establishes the connection between the specified target and Simulink.

Examples

Connect target to Simulink

Create a target object and add a target with name `LinuxTarget1`. Connect the target to Simulink.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");  
tg.connect();
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `disconnect`

Topics

“Set Up Linux Target Computer”

deployApplicationPackage

Deploy application package to selected target computer

Syntax

```
[status,applicationId] = deployApplicationPackage(tg, swPackageName)
```

Description

[status,applicationId] = deployApplicationPackage(tg, swPackageName) deploys the application package on to the target and returns the status.

Examples

Deploy Application Package to Target

Deploy the application package myApplicationPackage on to the target.

```
tg.deployApplicationPackage('myApplicationPackage')
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: myTargetHandle

swPackageName — Name of the application package

character vector | string scalar

Specify the name of the application package (mldatx file). You can also provide the path of the package if it is not available in the working folder.

Example: 'BrakeController', 'BrakeController.mldatx', '/matlab/autosar/BrakeController.mldatx'

Output Arguments

status — Status of application package deployment

logical

Returns `true` if application package is deployed on to the target. `false` if the application package cannot be deployed.

applicationId — Id of the deployed application package

string

Returns application Id of the application package if deployment is completed. And empty if deployment on to the target cannot be completed.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `linux.createApplicationPackage` |
`removeApplicationPackage`

Topics

“Build Simulink Model and Deploy Application”

disconnect

Disconnect Simulink from selected target computer

Syntax

```
disconnect(tg)
```

Description

`disconnect(tg)` removes the connection between the specified target and Simulink.

Examples

Disconnect Target From Simulink

Create a target object and add a target with name `LinuxTarget1`. Connect the target to Simulink.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");  
tg.connect();
```

Disconnect the target from Simulink.

```
tg.disconnect();
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `connect`

Topics

“Set Up Linux Target Computer”

getAllTargets

Get list of available target computers

Syntax

```
targetNames = getAllTargets(tgs)
```

Description

`targetNames = getAllTargets(tgs)` returns the list of the target names available in the target object.

Examples

Get the List of Available Targets

Create a target object and add targets with names `LinuxTarget1` and `LinuxTarget2`. Return the list of target names.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");  
tg1 = tgs.addTarget("LinuxTarget2", "178.10.10.2", "user1", "myPassword");  
targetNames = getAllTargets(tgs);
```

Input Arguments

tgs — Targets object

handle

Target object created by the `linux.Targets` class.

Example: `tgs`, `target_object`

Output Arguments

targetNames — List of the targets

array of strings

Contains the list of the target names that were added to the target object.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `getTarget`

Topics

“Set Up Linux Target Computer”

getApplications

Get applications deployed to target computer and application states

Syntax

```
stateMap = getApplications(tg)
```

Description

`stateMap = getApplications(tg)` returns a map where keys represent the names of the applications and the respective values indicate the state of the application on the target.

Examples

Get the Applications on Target

Get the list of applications deployed on target `LinuxTarget1` and their states.

```
stateMap = tg.getApplications();
```

The command returns `stateMap` in the form of a map, where keys are the names of applications deployed on target and values are the states of the applications.

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Output Arguments

stateMap — map containing name and state of the application

map

Returns a map in which keys are the names of the applications deployed on to the target. And values of the respective keys are the states of the applications on the target.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `getTarget`

Topics

“Start and Stop Deployed Applications”

getApplicationState

Get state of application deployed to target computer

Syntax

```
state = getApplicationState(tg, applicationName)
```

Description

`state = getApplicationState(tg, applicationName)` returns the state of the specific application on the particular target.

Examples

Get the Application State of Target

Get the state of deployed application state `myApplication` on the target.

```
tg.getApplicationState('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

applicationName — name of the application

character vector | string scalar

Specify the name of the application that is deployed on to the target.

Example: `myApplication`

Output Arguments

state — application state

enum

Returns an enum value indicating the state of the specific application deployed on to the target. State of the application can have following values.

State value	Indicating
IDLE	deployed application has not been launched or terminated

State value	Indicating
RUNNING	deployed application is running on the target
SUSPENDED	deployed application is paused on target

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `getApplications`

Topics

“Start and Stop Deployed Applications”

getCodeCoverageData

Extract coverage data for generated C/C++ code and custom C/C++ code (MATLAB code generation)

Syntax

```
coverageObject = getCodeCoverageData(myMEXFunction)
```

Description

`coverageObject = getCodeCoverageData(myMEXFunction)` generates C/C++ code coverage object by using data created by latest execution of `myMEXFunction` in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. Running this function also opens the C/C++ code coverage HTML report.

Examples

Add Instrumentation That Performs Coverage Analysis

This example shows how to use the `instrumentCode` function to add instrumentation for C/C++ coverage analysis to code that you already generated by using the `codegen` command. To perform coverage analysis, you need a MATLAB® Test™ license.

Define Entry-Point Function and Generate Code

Define a MATLAB entry-point function `coverageAnalysisExample` that performs a binary operation on its first two inputs. The binary operation depends on whether the third input is positive, negative, or zero.

```
type coverageAnalysisExample

function out = coverageAnalysisExample(A,B,flag) %#codegen
if flag > 0
    out = A + B;
elseif flag < 0
    out = A - B;
else
    out = A * B;
end
end
```

Generate static library code for software-in-the-loop (SIL) execution. Specify all inputs to be of scalar double type. Specify the name of the code generation folder to be `my_codegen_folder`.

```
cfg = coder.config('lib');
cfg.VerificationMode = 'SIL';
codegen -config cfg coverageAnalysisExample -args {0,0,0} -d my_codegen_folder
```

Code generation successful.

Add Instrumentation to Generated Code

Add C/C++ code coverage instrumentation to the generated code and produce an instrumented MEX.

```
instrumentCode('my_codegen_folder', 'CodeCoverage', true)
```

Completed instrumentation of generated code.

Execute the MEX and View Coverage Data

Execute coverageAnalysisExample_sil with two sets of sample inputs. Make sure the third input is positive for one set and negative for the other.

```
coverageAnalysisExample_sil(10,4,3)

### Starting SIL execution for 'coverageAnalysisExample'
To terminate execution: clear coverageAnalysisExample_sil

ans =
14

coverageAnalysisExample_sil(10,4,-2)

ans =
6
```

Terminate SIL execution and view the code coverage report.

```
clear coverageAnalysisExample_sil

### Application stopped
### Stopping SIL execution for 'coverageAnalysisExample'
Code coverage report: getCodeCoverageData('coverageAnalysisExample_sil')

getCodeCoverageData('coverageAnalysisExample_sil');
```

Code Coverage Report

The code coverage report provides a detailed analysis of the source code covered by the tests.

Overall Coverage Summary

Summary of the code coverage metrics for all source files.

Total Files

3

Coverage...	Executable	Missed	Code Co...
Function	3	0	100%
Statement	7	1	85.71%

Currently viewing:

Statement

Breakdown by Source

Code coverage metrics per source file.

Summary View Detailed View

Root Folder - C:\Users\aghosh\Documents\ExampleManager\aghosh.Bdoc23a.j2172582\ecoder-ex59520466\my_codegen_folder\

	File Name	Function	Statement
1	coverageAnalysisExample.c	100%	85.71%
2	coverageAnalysisExample_initialize.c	100%	N/A
3	coverageAnalysisExample_terminate.c	100%	N/A

Because neither of your test runs had the third input argument equal to zero, the last branch of the `if-else` block was never executed in the generated C entry-point file `coverageAnalysisExample.c`. This caused the `coverageAnalysisExample.c` file to have less than 100% statement coverage.

Input Arguments

myMEXFunction — Name of SIL or PIL MEX function

character vector | string scalar

Name of the software-in-the-loop (SIL) or processor-in-the-loop (PIL) MEX function for which you intend to generate C/C++ code coverage report.

Data Types: `char` | `string`

Version History

Introduced in R2023a

See Also

`codegen` | `coder.EmbeddedCodeConfig` | `instrumentCode`

getDeploymentLocation

Get deployment location for selected target computer

Syntax

```
location = getDeploymentLocation(tg)
```

Description

`location = getDeploymentLocation(tg)` returns the location where the software package is deployed on the target.

Examples

Get Deployment Location on Target

Get the location of deployment on the target using the following command:

```
tg.getDeploymentLocation();
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Output Arguments

location — folder location of the software package

character vector

Returns the folder path where the software package is deployed on the target.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `getTarget`

Topics

“Set Up Linux Target Computer”

“Build Simulink Model and Deploy Application”

getIPAddress

Get IP address of target computer

Syntax

```
ipAddress = getIPAddress(tg)
```

Description

`ipAddress = getIPAddress(tg)` returns the IP address of the target.

Examples

Get the IP address of the target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Get the IP address.

```
ipAddress = getIPAddress(tg);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Output Arguments

ipAddress — IP address of the target

string

Returns the IP address of the target.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `getTarget`

Topics

“Set Up Linux Target Computer”

“Build Simulink Model and Deploy Application”

getMeasurements

Get list of measurable signals

Syntax

```
signals = getMeasurements(tg, appName)
```

Description

`signals = getMeasurements(tg, appName)` returns the list of measurable signals on the application.

Examples

Get the List of Signals

Get the list of measurable signals for instrumentation of the application `myApplication`.

```
tg.getMeasurements('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to view the list of measurable signals.

Example: `myApplication`

Output Arguments

signals — list of measurable signals

array of strings

Returns the list of measurable signals on the application.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | [getParameters](#) | [getSignalsUnderInstrument](#)

Topics

[“Build Simulink Model and Deploy Application”](#)

[“Tune Parameters and Measure Signal Data for Deployed Applications”](#)

getParameters

Get list of tunable parameters

Syntax

```
parameters = getParameters(tg, appName)
```

Description

`parameters = getParameters(tg, appName)` returns the list of tunable parameters on the application.

Examples

Get the List of Parameters

Get the list of tunable parameters for calibration of the application `myApplication`.

```
tg.getParameters('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to view the list of tunable parameters.

Example: `myApplication`

Output Arguments

parameters — list of available parameters

array of strings

Returns the list of tunable parameters on the application.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | [getMeasurements](#) | [getSignalsUnderInstrument](#)

Topics

[“Tune Parameters and Measure Signal Data for Deployed Applications”](#)

getSignalsUnderInstrument

View signals selected to measure

Syntax

```
selectedSignals = getSignalsUnderInstrument(tg, appName)
```

Description

`selectedSignals = getSignalsUnderInstrument(tg, appName)` returns the list of signals that are selected to view for instrumentation in the application.

Examples

Get the Signals Being Measured

Get the list of signals being measured in the application `myApplication`.

```
tg.getSignalsUnderInstrument('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to view the list of measurable signals.

Example: `myApplication`

Output Arguments

selectedSignals — list of selected signals to instrument

array of strings

Returns the list of selected measurable signals to view on the application.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | [getMeasurements](#) | [getParameters](#)

Topics

[“Tune Parameters and Measure Signal Data for Deployed Applications”](#)

getSSHPort

Get port number connected to target computer

Syntax

```
PortNumber = getSSHPort(tg)
```

Description

`PortNumber = getSSHPort(tg)` returns the SSH port number connected to the target computer.

Examples

Get Connected Port Number

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Get the port number connected to the target computer.

```
portNumber = getSSHPort(tg);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Output Arguments

PortNumber — SSH port number

positive integer

Returns the SSH port number connected to the target computer.

Example: 22

Version History

Introduced in R2023b

See Also

setSSHPort | Linux Runtime Manager

Topics

“Set Up Linux Target Computer”

getTarget

Get handle of target computer

Syntax

```
tg = getTarget(tgs,targetName)
```

Description

`tg = getTarget(tgs,targetName)` returns the handle of the target `targetName` if it is already added to the target object.

Examples

Get the Handle of a Target

Add a target with name `LinuxTarget1` for the target object and get the handle of the target.

```
tgs = linux.Targets();  
addTarget(tgs, "LinuxTarget1", "178.10.10.1", "user1", "myPassword");  
tg = getTarget(tgs, "LinuxTarget1");
```

`tg` contains the handle for the target.

Input Arguments

tgs — Targets object

handle

Target object created by the `linux.Targets` class.

Example: `tgs, target_object`

targetName — Target name

character vector | string scalar

Specify a name for the target.

Example: `"LinuxTarget1"`

Output Arguments

tg — handle of the target

handle

Holds the handle for the specified target.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | connect | disconnect | deployApplicationPackage

Topics

"Set Up Linux Target Computer"

"Build Simulink Model and Deploy Application"

getTargetName

Get target computer name

Syntax

```
targetName = getTargetName(tg)
```

Description

targetName = getTargetName(tg) returns the name of the target.

Examples

Get the Target Name

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Get the name of the target.

```
targetName = getTargetName(tg);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Output Arguments

targetName — Name of the target

character vector | string scalar

Returns the name of the target.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `getUserName` | `getIPAddress`

Topics

“Set Up Linux Target Computer”

getUserName

Get user name of target computer

Syntax

```
userName = getUserName(tg)
```

Description

`userName = getUserName(tg)` returns the user name of the target computer.

Examples

Get the User Name

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Get the user name of the target computer.

```
userName = getUserName(tg);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Output Arguments

userName — Target computer user name

character vector | string scalar

Returns the user name of the target computer.

Version History

Introduced in R2022b

See Also

`getTargetName` | `Linux Runtime Manager` | `getIPAddress`

Topics

“Set Up Linux Target Computer”

isConnected

Get Boolean value representing target computer connection status

Syntax

```
status = isConnected(tg)
```

Description

`status = isConnected(tg)` returns the status as `true` if the target is connected. And `false` if the target is not connected.

Examples

Get the Status of Target Connection

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Get the status of the target connection.

```
status = isConnected(tg);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Output Arguments

status — Status of the target connection

logical

Status value will be set to `true` if the target is connected. And it will be set to `false` if the target is not connected.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | connect | disconnect

Topics

“Set Up Linux Target Computer”

pauseApplication

Pause application running on target computer

Syntax

```
pauseApplication(tg, appName)
```

Description

`pauseApplication(tg, appName)` suspends the specified application which is in running state.

Examples

Pause the Application

Pause the running application `myApplication`.

```
tg.pauseApplication('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to pause it.

Example: `myApplication`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `resumeApplication` | `startApplication` | `stopApplication`

Topics

“Start and Stop Deployed Applications”

readParameter

Get value of parameter

Syntax

```
value = readParameter(tg, appName,paramName)
```

Description

`value = readParameter(tg, appName,paramName)` returns the value of the specified parameter `paramName` in the application `appName`.

Examples

Get the Value of Parameter

Get the value of the parameter `myParameter` from the application `myApplication`.

```
val = tg.readParameter('myApplication','paramName');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name from where to get the value of parameter.

Example: `myApplication`

paramName — Name of the parameter

character vector | string scalar

Specify the parameter name to get the value.

Example: `myParameter`

Output Arguments

value — return value of the parameter

numeric | string | enum

Returns the value of the specified parameter from the deployed application.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `writeParameter` | `readParameters`

Topics

"Tune Parameters and Measure Signal Data for Deployed Applications"

readParameters

Get values of tunable parameters

Syntax

```
map = readParameters(tg, appName)
```

Description

`map = readParameters(tg, appName)` returns a map in which keys represent the parameters available in the application and values represent the corresponding values.

Examples

Read Parameters of an Application

Get the list of parameters and their values of the application `myApplication`.

```
tg.readParameters('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name from where to get the values of parameters..

Example: `myApplication`

Output Arguments

map — parameters and their values

map

Returns the map element in which the keys are the available parameters in the application and values in the map represent the values of the parameters.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `writeParameter` | `readParameter`

Topics

“Tune Parameters and Measure Signal Data for Deployed Applications”

removeSignalsFromInstrument

Remove signals for which visualization is not required

Syntax

```
removeSignalsFromInstrument(tg, appName, signals)
```

Description

`removeSignalsFromInstrument(tg, appName, signals)` removes the specified signals from the list of signals under visualization.

Examples

Remove Signals From Instrument

Remove the signals from measurement which are not intended to visualize on Simulink Data Inspector from the application `myApplication`.

```
tg.removeSignalsFromInstrument('myApplication',["signal_1","signal_2"]);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name.

Example: `myApplication`

signals — signals to be removed from instrument

array of strings

Specify the names of signals as an array which are not of interest to be visualized.

Example: `["signal1", "signal2"]`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | [selectSignalsToInstrument](#) | [startStreaming](#) | [stopStreaming](#)

Topics

[“Tune Parameters and Measure Signal Data for Deployed Applications”](#)

removeApplicationPackage

Remove application package from selected target computer

Syntax

```
removeApplicationPackage(tg, appName)
```

Description

`removeApplicationPackage(tg, appName)` removes the specified application and its artifacts from the target.

Examples

Remove Application From Target

Remove the deployed application package `myApplication` from the target.

```
tg.removeApplicationPackage('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to remove the application package.

Example: `myApplication`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `linux.createApplicationPackage` | `deployApplicationPackage`

Topics

“Build Simulink Model and Deploy Application”

removeTarget

Delete target computer

Syntax

```
removeTarget(tgs, targetName)
```

Description

`removeTarget(tgs, targetName)` deletes the specified target from the Targets Tree. This function does not remove the software or applications deployed on the target.

Examples

Remove Target From Target Object

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Remove the target.

```
removeTarget(tgs, "LinuxTarget1");
```

Input Arguments

tgs — Targets object

handle

Target object created by the `linux.Targets` class.

Example: `tgs, target_object`

targetName — name of a specific target

character vector | string scalar

Specify the name of the target which you want to delete from the Targets tree.

Example: `"LinuxTarget1"`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `addTarget` | `renameTarget`

Topics

“Set Up Linux Target Computer”

renameTarget

Rename target computer

Syntax

```
renameTarget(tgs, oldTargetName, newTargetName)
```

Description

`renameTarget(tgs, oldTargetName, newTargetName)` updates the name of the target from `oldTargetName` to `newTargetName`.

Examples

Rename the Target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Rename the target.

```
renameTarget(tgs, "LinuxTarget1", "LinuxTarget2");
```

Input Arguments

tgs — Targets object

handle

Target object created by the `linux.Targets` class.

Example: `tgs, target_object`

oldTargetName — Existing name of the target

character vector | string scalar

Specify the current name of the target to be updated.

Example: `"TargetName"`

newTargetName — New name of the target

character vector | string scalar

Specify the new name for the target.

Example: `"Target'sNewName"`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | addTarget | removeTarget

Topics

“Set Up Linux Target Computer”

restartStreaming

Restart streaming of signals

Syntax

```
restartStreaming(tg, appName)
```

Description

`restartStreaming(tg, appName)` updates the measurement to include or exclude the newly added or removed signals from the measurement of the specified application on the target and visualizes them using SDI.

Examples

Restart streaming of Signals

Restart measurement of updated signal set of the application `myApplication`.

```
tg.restartStreaming('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to restart the measurement.

Example: `myApplication`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `startStreaming` | `stopStreaming`

Topics

“Tune Parameters and Measure Signal Data for Deployed Applications”

resumeApplication

Resume application on target computer

Syntax

```
resumeApplication(tg, appName)
```

Description

`resumeApplication(tg, appName)` resumes the specified application after suspension.

Examples

Resume the Application

Resume the deployed application `myApplication` on the target.

```
tg.resumeApplication('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name.

Example: `myApplication`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `pauseApplication` | `startApplication` | `stopApplication`

Topics

“Start and Stop Deployed Applications”

selectSignalsToInstrument

Add signals of interest to visualize application behavior

Syntax

```
selectSignalsToInstrument(tg, appName, signals)
```

Description

`selectSignalsToInstrument(tg, appName, signals)` adds the specified signals to the list of signals under visualization.

Examples

Add Signals for Instrument

Add the signals of interest for measurement of the application `myApplication`.

```
tg.selectSignalsToInstrument('myApplication',["signal_1","signal_2"]);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name for calibration.

Example: `myApplication`

signals — signals to be added for instrument

array of strings

Specify the names of signals as an array which are of interest to be visualized.

Example: `["signal1", "signal2"]`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | removeSignalsFromInstrument | startStreaming | stopStreaming

Topics

“Tune Parameters and Measure Signal Data for Deployed Applications”

setApplicationStateCallback

Trigger a function when state of application changes

Syntax

setApplicationStateCallback(callback)

Description

setApplicationStateCallback(callback) calls the function provided in callback when the state of the specified application gets changed.

Examples

Set a Callback

Display the state transition for an application using the following function where appName is the name of the application, appState is the state of the application, and actionStatus is the status of an action performed on the application.

```
function applicationStateCallback(appName, appState, actionStatus)
if actionStatus
    switch appState
        case linux.AppState.IDLE
            disp(['application ' appName ' has moved to idle state.']);
        case linux.AppState.RUNNING
            disp(['application ' appName ' has moved to running state.']);
        case linux.AppState.SUSPENDED
            disp(['application ' appName ' has moved to suspended state.']);
    end
else
    disp(['Action performed on application ' appName ' has failed']);
end
end
tg.setApplicationStateCallback(@applicationStateCallback);
```

Input Arguments

callback — action to occur when state of application is changed

function

Provide a function with required action to occur when state of the application gets changed. A callback is expected to have 3 input arguments specifying application name, application state, and status of an action performed on the application.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | setTargetConnectionStatusCallback | startApplication

Topics

“Start and Stop Deployed Applications”

setDeploymentLocation

Set location for deployed application on target computer

Syntax

```
setDeploymentLocation(tg, locationOnTarget)
```

Description

`setDeploymentLocation(tg, locationOnTarget)` uses the specified location on target to persist the deployed applications.

Examples

Set Deployment Location on Target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Set the deployment location.

```
tg.setDeploymentLocation("/tmp/workingFolder");
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

locationOnTarget — Location on target for applications

string

Specify a location on target to persist the deployed application instead of placing them on a location on docker container. This helps in persisting the applications even when the docker crashes.

Example: `"/tmp/workingFolder"`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | [setIPAddress](#) | [setUserName](#) | [setPassword](#)

Topics

[“Start and Stop Deployed Applications”](#)

setIPAddress

Set IP address for target computer

Syntax

```
setIPAddress(tg, IPAddress)
```

Description

`setIPAddress(tg, IPAddress)` sets the specified IP address for the target.

Examples

Set IP Address For Target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Set the IP address.

```
tg.setIPAddress("178.12.121.150");
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

IPAddress — IP address for the target

character vector | string scalar

Specify the IP address for the selected target. Only IPv4 addresses are accepted.

Example: `"171.12.134.151"`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `setDeploymentLocation` | `setUserName` | `setPassword`

Topics

“Start and Stop Deployed Applications”

setPassword

Set password for target computer

Syntax

```
setPassword(tg, Password)
```

Description

`setPassword(tg, Password)` sets the specified password for the target.

Examples

Set Password For Target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Set the password.

```
tg.setPassword("myPassword");
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Password — password for the target

character vector | string scalar

Specify the password for the target.

Example: `"newPassword"`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `setDeploymentLocation` | `setIPAddress` | `setUserName`

Topics

“Start and Stop Deployed Applications”

setTargetConnectionStatusCallback

Set callback function to trigger when status of connection to target computer changes

Syntax

setTargetConnectionStatusCallback(callback)

Description

setTargetConnectionStatusCallback(callback) triggers a callback upon change in the connection status of a target computer.

Examples

Set Callback

Display the status of the target connection using the following function where **status** is the change in status of target connection.

```
function targetConnectionCallback(status)
if status
    disp('Target connected');
else
    disp('Target disconnected');
end
end
tg.setTargetConnectionStatusCallback(@targetConnectionCallback);
```

Input Arguments

callback — action to occur when target connection is changed

function

Provide a function with required action to occur when connection status of the target gets changed. The callback is expected to have 1 input argument representing the connection status.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | setApplicationStateCallback | startApplication

Topics

“Start and Stop Deployed Applications”

setUserName

Set user name for target computer

Syntax

```
setUserName(tg, Username)
```

Description

setUserName(tg, Username) sets the specified user name for the target computer.

Examples

Set User Name For Target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Set the user name.

```
tg.setUserName("myUserName");
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Username — User name for the target

character vector | string scalar

Specify the user name for the selected target.

Example: `"myUserName"`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `setDeploymentLocation` | `setIPAddress` | `setPassword`

Topics

“Start and Stop Deployed Applications”

setSSHPort

Set SSH port number

Syntax

```
setSSHPort(tg, Portnumber)
```

Description

`setSSHPort(tg, Portnumber)` sets the specified SSH port number to connect to the target computer.

Examples

Set Port Number

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Set the port number.

```
tg.setSSHPort(12345);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Portnumber — SSH Port number

positive integer

Specify the SSH port number to connect to the target computer. By default, port number 22 is used to connect to the target computer. Other valid port numbers that can be specified are from 1024 to 65535.

Example: `myTargetHandle`

Version History

Introduced in R2023b

See Also

getSSHPort | Linux Runtime Manager

Topics

“Set Up Linux Target Computer”

startApplication

Launch application

Syntax

```
startApplication(tg, appName)
```

Description

`startApplication(tg, appName)` starts running the specified application on the target.

Examples

Start the Application

Start running the application `myApplication`.

```
tg.startApplication('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to start.

Example: `myApplication`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `pauseApplication` | `resumeApplication` | `stopApplication`

Topics

“Start and Stop Deployed Applications”

startMonitorAndTune

Start parameter tuning and signal monitoring

Syntax

```
status = startMonitorAndTune(tg, appName)
```

Description

`status = startMonitorAndTune(tg, appName)` starts the calibration of the specified application on the target.

Examples

Start Calibration of an Application

Start calibration and measurement of the application `myApplication`.

```
tg.startMonitorAndTune('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to start the calibration.

Example: `myApplication`

Output Arguments

status — status of the calibration

logical

Returns `true` if the calibration of application has started without any errors. `false` if calibration does not start.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | [stopMonitorAndTune](#) | [startStreaming](#) | [stopStreaming](#)

Topics

[“Tune Parameters and Measure Signal Data for Deployed Applications”](#)

startStreaming

Start streaming of signals and visualize results

Syntax

```
status = startStreaming(tg, appName)
```

Description

`status = startStreaming(tg, appName)` starts streaming of signals of the specified application on the target and visualizes them using SDI.

Examples

Start Measurement of an Application

Start measurement of application `myApplication`.

```
tg.startStreaming('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to start the measurement.

Example: `myApplication`

Output Arguments

status — status of the measurement

logical

Returns `true` if the measurement of signals of the application has started. `false` if measurement does not start.

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | startMonitorAndTune | stopMonitorAndTune | stopStreaming

Topics

“Tune Parameters and Measure Signal Data for Deployed Applications”

stopApplication

Terminate application that is running on target computer

Syntax

```
stopApplication(tg, appName)
```

Description

`stopApplication(tg, appName)` terminates the specified application which is in running state.

Examples

Terminate the Application

Terminate the application `myApplication`.

```
tg.stopApplication('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name.

Example: `myApplication`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `pauseApplication` | `startApplication` | `resumeApplication`

Topics

“Start and Stop Deployed Applications”

stopMonitorAndTune

Stop parameter tuning of the application

Syntax

```
status = stopMonitorAndTune(tg, appName)
```

Description

`status = stopMonitorAndTune(tg, appName)` stops the calibration of the specified application on the target.

Examples

Stop Calibration of an Application

Stop measurement and calibration of application `myApplication`.

```
tg.stopMonitorAndTune('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to stop the calibration.

Example: `myApplication`

Output Arguments

status — status of the calibration

logical

Returns `true` if the calibration of application has stopped. `false` if calibration does not stop.

Version History

Introduced in R2022b

stopStreaming

Stop streaming signal data

Syntax

```
status = stopStreaming(tg, appName)
```

Description

`status = stopStreaming(tg, appName)` stops streaming of signals of the specified application on the target.

Examples

Stop Measurement of Signals

Stop measurement of application `myApplication`.

```
tg.stopStreaming('myApplication');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to stop the measurement.

Example: `myApplication`

Output Arguments

status — status of the measurement

logical

Returns `true` if the measurement of signals of the application has stopped. `false` if measurement does not stop.

Version History

Introduced in R2022b

See Also

[startStreaming](#) | [Linux Runtime Manager](#) | [startMonitorAndTune](#) | [stopMonitorAndTune](#)

Topics

[“Tune Parameters and Measure Signal Data for Deployed Applications”](#)

uninstallTarget

Delete docker image on target computer

Syntax

```
uninstallTarget(tg)
```

Description

`uninstallTarget(tg)` deletes the docker image, docker container, and deployed applications on the target.

Examples

Uninstall Target

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Uninstall the target.

```
tg.uninstallTarget();
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `updateTarget` | `addTarget` | `removeTarget`

Topics

"Set Up Linux Target Computer"

updateTarget

Update docker image on target computer if there is a version mismatch

Syntax

```
updateTarget(tg)
```

Description

`updateTarget(tg)` updates the docker image on the target if there is a version mismatch.

Examples

Update Target Software

Create a target object and add a target with name `LinuxTarget1` containing user name `user1` and password `myPassword`.

```
tgs = linux.Targets();  
tg = tgs.addTarget("LinuxTarget1", "178.10.10.1", "user1", "myPassword");
```

Update docker image on target.

```
tg.updateTarget();
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `addTarget` | `removeTarget` | `uninstallTarget`

Topics

“Set Up Linux Target Computer”

coder.updateTlcForLanguageStandardTypes

Modify TLC file containing hard-coded Simulink Coder data types for compatibility with `DataTypeReplacement=CDataTypesFixedWidth` code generation

Syntax

```
coder.updateTlcForLanguageStandardTypes(tlcFile)
```

Description

`coder.updateTlcForLanguageStandardTypes(tlcFile)` updates the input TLC file, which contains hard-coded instances of Simulink Coder data types, for compatibility with `DataTypeReplacement=CDataTypesFixedWidth` code generation. In the TLC file, the function replaces the hard-coded instances of `CoderTypedef` names (for example, `real32_T`) with corresponding API calls. During code generation, the updated TLC code emits:

- C99 data types if the value of the `DataTypeReplacement` configuration parameter is `'CDataTypesFixedWidth'`.
- Simulink Coder data types if the `DataTypeReplacement` configuration parameter is `'CoderTypedefs'`.

The function does not update hard-coded data types that are used as a `case` value within a `switch` statement. In this instance, manually update the lines of code. For example, in the `switch` expression, use the data type identifier instead of the data type name.

Note Before generating code, in the modified TLC file, check the changes produced by the function. TLC files can be very complex. In some cases, the function updates might fail or introduce an error.

Examples

Update TLC File for Compatibility with `DataTypeReplacement=CDataTypesFixedWidth` Code Generation

This example shows how you can use `coder.updateTlcForLanguageStandardTypes` to update a TLC file that contains hard-coded instances of Simulink Coder data types.

Copy the TLC file to your working folder. For example, copy the `sldemo_sfun_dlut3D.tlc` file from a release before R2023a.

```
matlabRootForR2021b='C:\Program Files\MATLAB\R2021b';  
tlcFilePath=['\examples\simulink_features' ...  
            '\data\sldemo_sfun_dlut3D.tlc'];  
copyfile([matlabRootForR2021b tlcFilePath], '.');
```

Make the file writable.

```
fileattrib('sldemo_sfun_dlut3D.tlc', '+w');
```


Run the function.

```
coder.updateTlcForLanguageStandardTypes('sldemo_sfun_dlut3D.tlc');
```

File

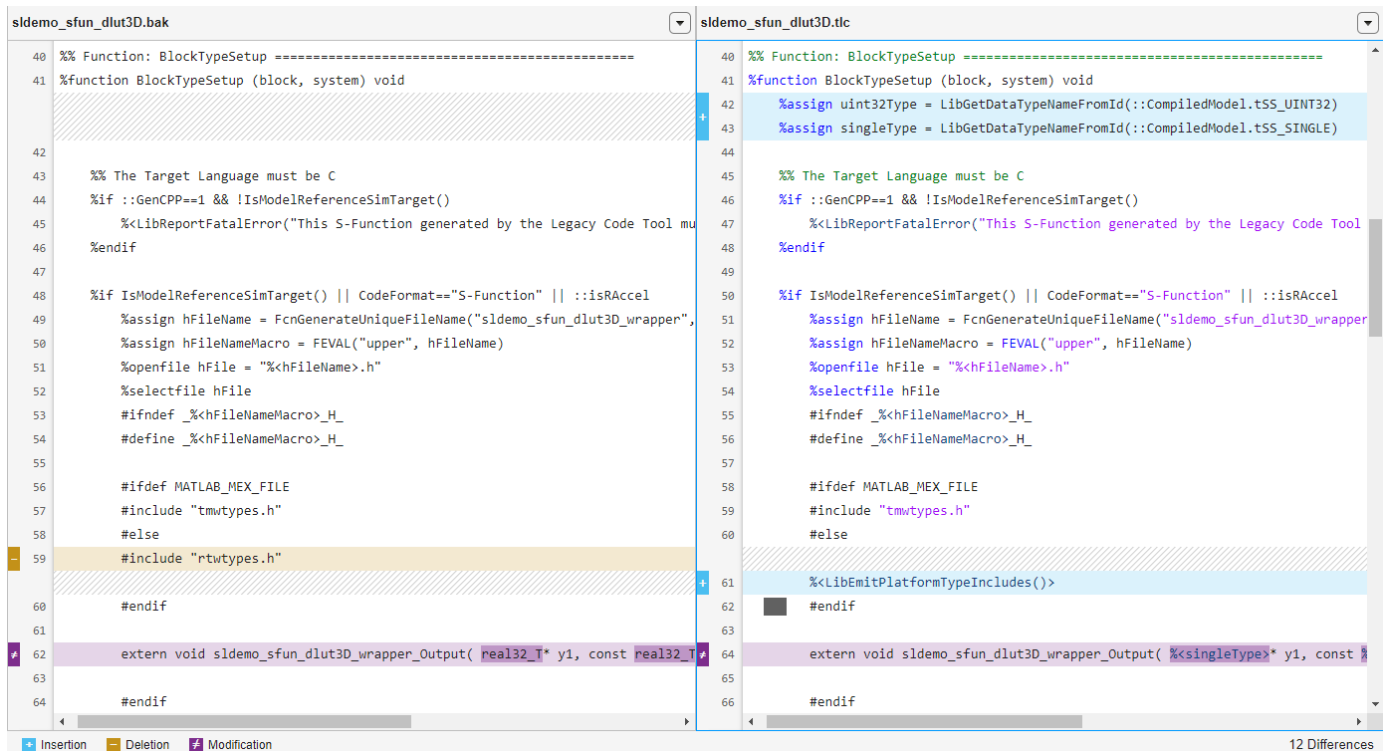
sldemo_sfun_dlut3D.tlc

was updated to support language standard types
Created backup file:

sldemo_sfun_dlut3D.bak

Please review differences between the two files.

Click the [review differences](#) link. The Comparison Tool displays the differences between the updated file and the backup file that contains the original code.



Observe, for example, that the call to the `singleType` function replaces the Simulink Coder data type `real32_T`.

Input Arguments

tlcFile — TLC filename

character vector | string scalar

Name of TLC file that contains hard-coded instances of Simulink Coder data types.

Example: 'sldemo_sfun_dlut3D.tlc'

Data Types: char | string

Version History

Introduced in R2023a

See Also

Data type replacement

Topics

“Compatibility of TLC Files with Generated Code Data Types”

“Manage Replacement of Simulink Data Types in Generated Code”

writeParameter

Write or tune application parameter

Syntax

```
writeParameter(tg, appName, paramName, paramVal)
```

Description

`writeParameter(tg, appName, paramName, paramVal)` writes or tunes the specified parameter on an application.

Examples

Tune a Parameter

Change the value of the parameter `myParameter` to the value `myValue` of the application `myApplication`.

```
tg.writeParameter('myApplication','myParameter','myValue');
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

appName — Name of the application

character vector | string scalar

Specify the application name to tune a parameter.

Example: `myApplication`

paramName — Name of the parameter

character vector | string scalar

Specify the name of the parameter to tune.

Example: `myParameter`

paramVal — Value of the parameter

double | integer | boolean | enum | structure | fixed

Specify a value for the parameter. The value field supports multiple data types to tune various kinds of parameters.

Example: 0.2

Example: 21

Version History

Introduced in R2022b

See Also

Linux Runtime Manager | `readParameter` | `startStreaming`

Topics

“Analyze Logged Messages”

Linux Runtime Manager

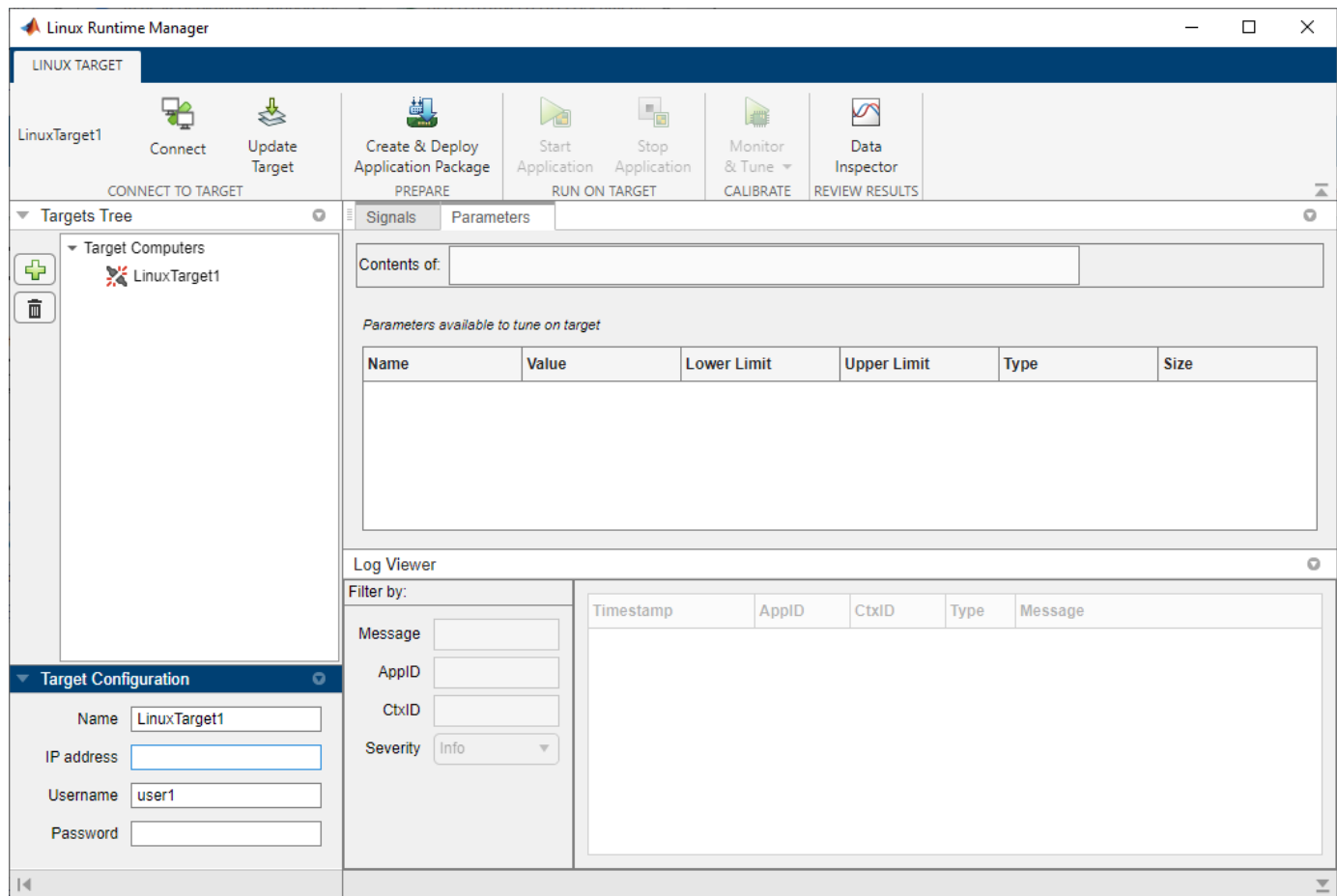
Deploy and prototype service-oriented architecture (SOA) application components

Description

The Linux Runtime Manager application provides a single point of contact for viewing connection status and interacting with Linux target computers. You can deploy the software components as a standalone Linux applications, display the log messages, monitor and trace signals, tune parameters, and stream data to the Simulation Data Inspector.

Use the Linux Runtime Manager application for these tasks:

- Connect the development computer and multiple Linux targets.
- Create and deploy the software component as standalone Linux application onto the Linux target.
- Start, and stop an application on target computer.
- Display and filter the log messages.
- View the list of available signals and parameters in the application.
- Measure the signals in the Simulation Data Inspector.
- Tune the parameters.
- Start and stop streaming of signals.
- Update the docker container on target Linux machine.
- Configure the target Linux machine for deployment.



Open the Linux Runtime Manager App

- At the MATLAB Command prompt: Enter
`linuxRuntimeManager`

Examples

- “Set Up Linux Target Computer”

Limitations

- **Password** field in the **Target Configuration** section on Linux Runtime Manager application does not get updated when other target is selected.
- If the docker container of a connected target goes down, it gets disconnected. But the Linux Runtime Manager application fails to update the status of connection.

Version History

Introduced in R2022b

Topics

“Set Up Linux Target Computer”

linux.utils.migrateDds2Adaptive

Convert DDS blockset model to AUTOSAR Adaptive model

Syntax

```
linux.utils.migrateDds2Adaptive(DDS_model_name)
linux.utils.migrateDds2Adaptive(DDS_model_name,Name=Value)
```

Description

`linux.utils.migrateDds2Adaptive(DDS_model_name)` converts the DDS blockset model `DDS_model_name` to AUTOSAR Adaptive model and saves it as `DDS_model_name_adaptive` in the current working directory. It also creates the `<DDSdict_name>_adaptive.sldd` data dictionary.

`linux.utils.migrateDds2Adaptive(DDS_model_name,Name=Value)` converts the DDS blockset model `DDS_model_name` to AUTOSAR Adaptive model with one or more "Name=Value" arguments. For example, you can specify a name to save the converted model.

Examples

Convert DDS Model to AUTOSAR Adaptive Model

Open a DDS Blockset model.

Use the command to convert the model.

```
[adaptiveModelName, adaptiveDictName]=linux.utils.migrateDds2Adaptive(<modelName>)
```

```
adaptiveModelName =
'<modelName>_adaptive'
adaptiveDictName =
'<slddName>_adaptive.sldd'
```

You can see an AUTOSAR Adaptive model and Simulink® data dictionary gets created in the same folder.

Input Arguments

DDS_model_name — Name of the DDS blockset model

char vector | string

Specify the name of the DDS blockset model to be converted to AUTOSAR Adaptive model.

Example: `my_dds_model`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `linux.utils.migrateDds2Adaptive('ex_positionestimator', AdaptiveModelName='ex_positionestimator_adaptive', DictionaryName='adaptive_data_dictionary')` converts the `ex_positionestimator` model into AUTOSAR Adaptive model and saves it as `ex_positionestimator_adaptive` and creates a new data dictionary for Adaptive version of the model and save it as `adaptive_data_dictionary`.

AdaptiveModelName — Name for the AUTOSAR Adaptive model

char vector | string

Specify a name to save the converted model.

Example: `AdaptiveModelName='my_adaptive_model'`

DictionaryName — Name for the data dictionary

char vector | string

Specify a name to save the data dictionary for the converted model.

Example: `DictionaryName='adaptive_data_dict'`

Tips

DDS Blockset models uses broadcasting way of communication and AUTOSAR adaptive models uses locator based communication. In order to establish communication between AUTOSAR adaptive and DDS Blockset models,

- Set the **Code Generation > Build process > Build configuration** to Specify.
- Specify the **Code Generation > Build process > Toolchain details > Defines** as - `DDO_NOT_USE_LOCATORS`.

Version History

Introduced in R2022b

See Also

Topics

“Video - Convert DDS Blockset Model to AUTOSAR Adaptive Model”
 “Build Simulink Model and Deploy Application”

linux.LogMessages

Create object that contains target log messages

Syntax

```
logObj=linux.LogMessages(targetName)
```

Description

`logObj=linux.LogMessages(targetName)` creates an object `logObj` containing the log messages of the applications running on the specified target `targetName`.

Examples

Create a Log Object

Create a log object with name `log_Obj_Target1` for the Target1.

```
log_Obj_Target1 = linux.LogMessages('Target1');
```

Input Arguments

targetName — Name of the target

char vector | string

Specify the name of the target.

Example: `LinuxTarget1`

Output Arguments

logObj — Object containing log messages

object

Returns an object containing the log messages of the applications running on the specified target.

Version History

Introduced in R2022b

See Also

`linux.LogMessages.get`

Topics

“Analyze Logged Messages”

linux.LogMessages.get

Filter and get log messages

Syntax

```
get(logObj)
```

Description

`get(logObj)` returns the log messages from the log object `logObj` filtered based on application Id `appIdFilter`, context Id `ctxIdFilter`, severity level `severityFilter`, and containing the text `messageFilter`.

Examples

Filter the Log Messages

Create a log object with name `log_Obj_Target1` for the `Target1`.

```
log_Obj_Target1 = linux.LogMessages('Target1');
```

From the `log_Obj_Target1`, view the messages related to the application `Linux Application1`.

```
get(log_Obj_Target1, AppIdFilter='Linux_Application1')
```

From the `log_Obj_Target1`, view the messages with severity level `Warn` and related to the application `Linux Application2`.

```
get(log_Obj_Target1, AppIdFilter='Linux_Application1', SeverityFilter='Warn')
```

Input Arguments

logObj — object containing the log messages

object

Specify the object containing logs of a target. The `logObj` can be created by using the `linux.LogMessages` function.

Example: `log_Object1`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `get(logObj, AppIdFilter='Linux_application1', CtxIdFilter='Linux application', SeverityFilter='Warn')` returns the log messages related to the `Linux_application1`, `Linux` application context, and with severity `Warn` and above from the `logObj`.

MessageFilter — Filter by message text

`char vector` | `string`

Filter the log messages by a string. Specify the string you want to see in the log messages.

Example: `MessageFilter="User Defined Binding"`

AppIdFilter — Filter by application id

`char vector` | `string`

Filter the log messages by unique application identifier. Specify the application id to see the log messages related to that application.

Example: `AppIdFilter='Linux_Application1'`

CtxIdFilter — Filter by message context

`char vector` | `string`

Filter the log messages by context identifier. Specify the context id to see in the log messages. Context id can be same in multiple applications. To view log messages of specific context from a specific application, use both `AppIdFilter` and `CtxIdFilter`.

Example: `CtxIdFilter='Linux App'`

Example: `AppIdFilter='Linux_Application1',CtxIdFilter='Linux App'`

SeverityFilter — Filter by severity level

`char vector` | `string`

Filter the log messages by the level of severity. Specify the severity level to view the messages with specified severity and above. The order of the severity is as follows:

- 1** Fatal
- 2** Error
- 3** Warn
- 4** Info
- 5** Debug
- 6** Verbose

Example: `SeverityFilter='Error'`

Version History

Introduced in R2022b

See Also

`linux.LogMessages`

Topics

“Analyze Logged Messages”

linuxTarget

Interface for managing target computer

Description

`linuxTarget` constructs a target object which represents a target computer and provides access to methods and properties related to the target computer.

The object provides access to methods and properties that:

- Start, pause, resume, and stop the application.
- Read and set parameters.
- Monitor signals.
- Retrieve status information about the target computer.
- Restart the docker container on target computer.
- Load and deploy the application.

Creation

`tg = linuxTarget` constructs a target object representing the only available target computer.

`tg = linuxTarget(targetName)` constructs a target object representing the target computer designated by `targetName`.

Example: “Only One Target Computer Present” on page 1-725

Example: “Multiple Target Computers Present” on page 1-725

Object Functions

<code>connect</code>	Connect Simulink to selected target computer
<code>deployApplicationPackage</code>	Deploy application package to selected target computer
<code>disconnect</code>	Disconnect Simulink from selected target computer
<code>getApplicationState</code>	Get state of application deployed to target computer
<code>getApplications</code>	Get applications deployed to target computer and application states
<code>getDeploymentLocation</code>	Get deployment location for selected target computer
<code>getIPAddress</code>	Get IP address of target computer
<code>getMeasurements</code>	Get list of measurable signals
<code>getParameters</code>	Get list of tunable parameters
<code>getSignalsUnderInstrument</code>	View signals selected to measure
<code>getTargetName</code>	Get target computer name
<code>getUserName</code>	Get user name of target computer
<code>isConnected</code>	Get Boolean value representing target computer connection status
<code>pauseApplication</code>	Pause application running on target computer

readParameter	Get value of parameter
readParameters	Get values of tunable parameters
removeApplicationPackage	Remove application package from selected target computer
removeSignalsFromInstrument	Remove signals for which visualization is not required
restartContainer	Restart or recreate docker container
restartStreaming	Restart streaming of signals
resumeApplication	Resume application on target computer
selectSignalsToInstrument	Add signals of interest to visualize application behavior
setApplicationStateCallback	Trigger a function when state of application changes
setDeploymentLocation	Set location for deployed application on target computer
setIPAddress	Set IP address for target computer
setPassword	Set password for target computer
setTargetConnectionStatusCallback	Set callback function to trigger when status of connection to target computer changes
setUserName	Set user name for target computer
startApplication	Launch application
startMonitorAndTune	Start parameter tuning and signal monitoring
startStreaming	Start streaming of signals and visualize results
stopApplication	Terminate application that is running on target computer
stopMonitorAndTune	Stop parameter tuning of the application
stopStreaming	Stop streaming signal data
uninstallTarget	Delete docker image on target computer
updateTarget	Update docker image on target computer if there is a version mismatch
writeParameter	Write or tune application parameter

Examples

Only One Target Computer Present

Create a target object that communicates with the available target computer. Select the available target computer by using the `linuxTarget`.

```
tg=linuxTarget;
```

Multiple Target Computers Present

Create a target object that communicates with the specified target computer. Select a specific target computer by using the `linuxTarget`.

```
tg=linuxTarget("Target_1");
```

Version History

Introduced in R2024a

See Also

`addTarget` | `getAllTargets`

Topics

"Set Up Linux Target Computer"

"Tune Parameters and Measure Signal Data for Deployed Applications"

restartContainer

Restart or recreate docker container

Syntax

```
restartContainer(tg, recreateContainer)
```

Description

`restartContainer(tg, recreateContainer)` restart or recreate the docker container on target machine from host machine.

Examples

Restart or Recreate Docker Container

Get the target on which the docker container has to be restarted.

```
tg = linuxTarget("<Target_Name>");
```

Restart the docker container on target computer from host machine.

```
tg.restartContainer;
```

Recreate the docker container on target computer from host machine.

```
tg.restartContainer(recreateContainer=true);
```

Input Arguments

tg — Object representing a target computer

handle

Object represents a target computer. Use the object to establish the connection with Linux target, to deploy the applications on Linux target, and perform several other actions.

Example: `myTargetHandle`

recreateContainer — Recreate docker container

false (default) | true

Set the `recreateContainer` to `true` to recreate the docker container on target computer from host machine. Set the `recreateContainer` to `false` to restart the docker container.

Example: `recreateContainer=true`

Version History

Introduced in R2024a

See Also

Linux Runtime Manager

Topics

“Start and Stop Deployed Applications”

linux.utils.getApplicationPackageDescription

Get application package description

Syntax

```
pkgDesc=linux.utils.getApplicationPackageDescription(appPackageName)
```

Description

`pkgDesc=linux.utils.getApplicationPackageDescription(appPackageName)` returns the `pkgDesc` variable containing the description of application package such as name, version, type, whether the application contains executable or source code, and type of architecture for which the application package deployment is supported. The architecture field displays `x86_64` for Linux targets and `arm64` for ARM targets.

Examples

Get Application Description

Get the application package details for myApplication.

```
pkgDesc = linux.utils.getApplicationPackageDescription("myApplication.mldatx");
```

Input Arguments

appPackageName — Name of the application package

character vector | string scalar

Specify the application package name or full path for the application package.

Example: myApplication

Output Arguments

pkgDesc — Structure variable

structure

Structure variable containing the name, version, and type of the application package.

Version History

Introduced in R2024a

See Also

Linux Runtime Manager | `startApplication` | `pauseApplication` | `resumeApplication` | `stopApplication`

Topics

“Start and Stop Deployed Applications”

coder.profile.test.analyzePath

Namespace: coder.profile.test

Identify critical paths of tasks

Syntax

```
resultsObject = coder.profile.test.analyzePath(modelName, SimMode="SIL")
resultsObject= coder.profile.test.analyzePath(modelName,
TestFile="myTestFile")
resultsObject= coder.profile.test.analyzePath(..., Name=Value)
```

Description

`resultsObject = coder.profile.test.analyzePath(modelName, SimMode="SIL")` collects execution-time metrics for tasks through a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation using the model inputs. Then, it analyzes the metrics and identifies the code execution paths of the tasks that take the longest measured execution times, which are the critical paths.

The function stores the results in the `resultsObject` object. The results include:

- Identified critical paths.
- Estimated cost percentages of code sections in the identified paths.

Use Code Profile Analyzer to analyze the results in `resultsObject`.

`resultsObject= coder.profile.test.analyzePath(modelName, TestFile="myTestFile")` uses the model's **SIL/PIL Mode** and test cases in the Simulink Test file `myTestFile` for the analysis.

`resultsObject= coder.profile.test.analyzePath(..., Name=Value)` specifies additional name-value arguments.

Examples

Use coder.profile.test.analyzePath to Identify Critical Paths

You can use `coder.profile.test.analyzePath` and the Code Profile Analyzer to identify critical paths of the generated tasks and performance bottlenecks in them.

For an example, see “Identify Critical Path of Generated Code”.

Input Arguments

modelName — Name of model

character vector | string

Name of the model, specified as a character vector or string.

Example: "myModel"

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `resultsObject = coder.profile.test.analyzePath("myModel", SimMode="SIL")`

InstrumentationMode — Code segment size for instrumentation

"FineGrained" (default) | "CoarseGrained"

Types of instrumentation approach for the analysis:

- "FineGrained": Instruments each decision branch.
- "CoarseGrained": Instruments tasks.

Example: `resultsObject = coder.profile.test.analyzePath("myModel", InstrumentationMode="CoarseGrained", SimMode="SIL")`

Data Types: `char` | `string`

TestFile — MLDATX test file path

character vector | `string`

`TestFile` specifies the path to the MLDATX test file. When you specify the path to the test file, the `coder.profile.test.analyzePath` function uses the model's **SIL/PIL Mode** and the test cases in the file for the analysis. If you want to specify the simulation mode, set a value for `SimMode`.

Example: `resultsObject = coder.profile.test.analyzePath("myModel", TestFile="New Folder\tapobj\slsv_output\mymodel\mymodel_test.mldatx")`

Data Types: `char` | `string`

SimMode — Simulation mode

`SIL` | `PIL`

Simulation mode specified as `SIL` or `PIL`. If you want to use model inputs for the analysis, you must specify a simulation mode.

Example: `resultsObject = coder.profile.test.analyzePath("myModel", SimMode="SIL")`

Data Types: `char` | `string`

Verbose — Verbosity

`false` (default) | `true`

When you set `Verbose` to `true`, the function displays diagnostic messages after each step of the analysis in the MATLAB Command Window. If the function encounters errors, it generates a PDF file in your working directory providing information on the root cause of the errors.

Example: `resultsObject = coder.profile.test.analyzePath("myModel", SimMode="SIL", Verbose=true)`

Data Types: `logical`

Output Arguments

resultsObject — Result object

`coder.profile.ExecutionTimeSet` object

This object stores the identified critical paths and estimated cost percentages of code sections in the identified paths.

Use Code Profile Analyzer to analyze the results. To get a cell array of model elements in the identified critical path, use the function.

Version History

Introduced in R2024a

See Also

`coder.profile.test.generateTests`

Topics

“Analyze Execution Times of Critical Paths”

`coder.profile.test.generateTests`

Namespace: `coder.profile.test`

Generate decision coverage tests for execution-time analysis

Syntax

```
decisionCoverageTest = coder.profile.test.generateTests(modelName)  
decisionCoverageTest = coder.profile.test.generateTests(modelName,Name=Value)
```

Description

`decisionCoverageTest = coder.profile.test.generateTests(modelName)` generates decision coverage test cases for the specified model by using Simulink Design Verifier™. The function ensures 100% decision coverage. It saves the generated test cases in both a MAT data file and a Simulink Test compatible MLDATX file. It also generates an harness to run the test file in the current folder.

If you have a Simulink Test license, you can use the MLDATX file for execution-time analysis. If you do not have a Simulink Test license, you can import the tests in the MAT data file to your model as inputs and use those for analysis. For more information on importing data into a model, see “Overview of Signal Loading Techniques” and “Load Input Data for Basic Test Cases”. To generate test cases, this function can change model parameter settings which are restored back to the original settings after test generation. In such cases, you get a warning message.

`decisionCoverageTest = coder.profile.test.generateTests(modelName,Name=Value)` specifies additional name-value arguments. Use this syntax with the input argument in the preceding syntax.

Examples

Use `coder.profile.test.generateTests` to Generate Decision Coverage Tests

This example show how you can use `coder.profile.test.generateTests` to generate decision coverage tests.

Open the `CriticalPathIdentify` model.

```
openExample('CriticalPathIdentify');  
model= "CriticalPathIdentify";
```

Generate decision coverage tests, using this command:

```
decisionCoverageTest = coder.profile.test.generateTests(model)
```

Data file:
 \users\taprij\sldv_output\CriticalPathIdentify\CriticalPathIdentify_sldvdata.mat

```
decisionCoverageTest =
```



```
'\users\taprij\sldv_output\CriticalPathIdentify\CriticalPathIdentify_test.mldatx'
```

Input Arguments

modelName — Name of model

character vector | string

Name of the model, specified as a character vector or string.

Example: "CriticalPathIdentify"

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `decisionCoverageTest = coder.profile.test.generateTests("mymodel", MaxTime=100)`

options — Design verification options object

sldvoptions object

You can provide a custom `sldvoptions` object. If you do not provide an object, the function creates an object that have these parameter settings, which are different from their default settings.

Parameter	Value
TestgenTarget	'GenCodeTopModel'
RandomizeNoEffectData	'on'
ModelCoverageObjectives	'Decision'

For more information, see `sldvoptions`.

MaxTime — Maximum time for test generation

positive integer

Maximum time for test generation, specified as a positive integer.

`MaxTime` specifies the maximum time in seconds for test generation. If the specified time is not sufficient for test generation, the function issues an error and suggests increasing `MaxTime` value.

Example: `decisionCoverageTest = coder.profile.test.generateTests("mymodel", MaxTime=100)`

Verbose — Option to display test generation progress details

false (default) | true

When you set `Verbose` to `true`, the function displays the test generation progress details in the MATLAB Command Window and opens the Simulink Design Verifier Results Summary window.

Example: `decisionCoverageTest = coder.profile.test.generateTests("mymodel", Verbose=true)`

Data Types: `logical`

Output Arguments

decisionCoverageTest — Path of MLDATX test file

character vector

Path of the MLDATX test file, returned as a character vector. The function also returns a path to the MAT data file containing the generated test cases.

Version History

Introduced in R2024a

See Also

`coder.profile.test.analyzePath` | `coder.profile.test.generateCriticalPathTest`

Topics

“Analyze Execution Times of Critical Paths”

coder.profile.test.getCriticalPathElements

Namespace: coder.profile.test

Get cell array of model elements in critical path

Syntax

```
criticalPathBlocks = coder.profile.test.getCriticalPathElements(
resultsObject)
```

Description

`criticalPathBlocks = coder.profile.test.getCriticalPathElements(resultsObject)` returns a cell array of blocks that are in the critical path identified by `coder.profile.test.analyzePath`.

Examples

Get Blocks in Critical Path

Use `coder.profile.test.getCriticalPathElements` to get a cell array of blocks in the identified critical path.

Open the `CriticalPathIdentify` model.

```
openExample('CriticalPathIdentify');
model = "CriticalPathIdentify";
```

Identify the critical path of the generated code by using the `coder.profile.test.analyzePath` function and store the results in `myResults`.

```
myResults = coder.profile.test.analyzePath(model,SimMode="SIL")
```

Get a list of the blocks in the identified path.

```
criticalPathBlocks = coder.profile.test.getCriticalPathElements(myResults)
```

```
criticalPathBlocks =
```

24×1 cell array

```
{'CriticalPathIdentify/If' }
{'CriticalPathIdentify/If Action1' }
{'CriticalPathIdentify/If Action1/Action Port' }
{'CriticalPathIdentify/If Action1/For Iterator1' }
{'CriticalPathIdentify/If Action1/For Iterator1/For Iterator' }
{'CriticalPathIdentify/If Action1/For Iterator1/Sum' }
{'CriticalPathIdentify/If Action1/For Iterator1/Unit Delay' }
{'CriticalPathIdentify/If Action1/In1' }
{'CriticalPathIdentify/If Action1/Out1' }
{'CriticalPathIdentify/If Action2' }
```

```
{'CriticalPathIdentify/If Action2/Action Port'           }  
{'CriticalPathIdentify/If Action2/For Iterator2'        }  
{'CriticalPathIdentify/If Action2/For Iterator2/For Iterator'}  
{'CriticalPathIdentify/If Action2/For Iterator2/Sum'     }  
{'CriticalPathIdentify/If Action2/For Iterator2/Unit Delay'}  
{'CriticalPathIdentify/If Action2/In1'                   }  
{'CriticalPathIdentify/If Action2/Out1'                   }  
{'CriticalPathIdentify/If1'                               }  
{'CriticalPathIdentify/In1'                               }  
{'CriticalPathIdentify/In2'                               }  
{'CriticalPathIdentify/In4'                               }  
{'CriticalPathIdentify/Merge'                             }  
{'CriticalPathIdentify/Merge1'                             }  
{'CriticalPathIdentify/Out1'                               }
```

Input Arguments

resultsObject — Critical path analysis result

`coder.profile.ExecutionTimeSet` object

Critical path analysis result, specified as a `coder.profile.ExecutionTimeSet` object. This object is returned by the `coder.profile.test.analyzePath` function

Output Arguments

criticalPathBlocks — Blocks in critical path

cell array

Blocks in the critical path, specified as a cell array.

Version History

Introduced in R2024a

See Also

`coder.profile.test.analyzePath`

Topics

“Analyze Execution Times of Critical Paths”

coder.profile.test.generateCriticalPathTest

Namespace: coder.profile.test

Generate tests for critical paths

Syntax

```
pathCoverageTest = coder.profile.test.generateCriticalPathTest(resultsObject,  
MaxTime=100)
```

Description

`pathCoverageTest = coder.profile.test.generateCriticalPathTest(resultsObject, MaxTime=100)` generates test cases that execute the critical path of the generated code identified by `coder.profile.test.analyzePath` using Simulink Design Verifier. The function uses the maximum time specified in `MaxTime` for test generation. The function can generate test cases only if the path contains conditional subsystems or blocks with decision conditions for execution from this list.

- Switch
- If Action Subsystem
- Multiport Switch
- Switch Case
- For Iterator
- While Iterator
- Enabled Subsystem
- Triggered Subsystem
- Enabled and Triggered Subsystem

The `coder.profile.test.generateCriticalPathTest` function saves the generated test cases in both a MAT data file and a Simulink Test compatible MLDATX file. The files are saved in `sldv_output/modelName` folder, where `modelName` is the name of your model.

If you have a Simulink Test license, you can use the MLDATX file directly for execution-time analysis. If you do not have a Simulink Test license, you can import the test cases in the MAT data file to your model as inputs and use those for analysis. For more information on importing data into a model, see “Overview of Signal Loading Techniques” and “Load Input Data for Basic Test Cases”.

Examples

Generate Path Coverage Tests for Critical Paths

Use `coder.profile.test.generateCriticalPathTest` to generate tests for critical paths.

Open the `CriticalPathIdentify` model.

```
openExample('CriticalPathIdentify');  
model = "CriticalPathIdentify";
```

Identify the critical path of the generated code by using the `coder.profile.test.analyzePath` function and store the results in `myResults`.

```
myResults = coder.profile.test.analyzePath(model,SimMode="SIL")
```

Generate tests for the identified path using this command:

```
myTestFile = coder.profile.test.generateCriticalPathTest(myResults)
```

```
Data file:  
/users/sldv_output/CriticalPathIdentify/CriticalPathIdentify_sldvdata.mat
```

```
myTestFile =  
  
'/users/sldv_output/CriticalPathIdentify/CriticalPathIdentify_test.mldatx'
```

Input Arguments

resultsObject — Critical path analysis result

`coder.profile.ExecutionTimeSet` object

Critical path analysis result, specified as a `coder.profile.ExecutionTimeSet` object. This object is returned by the `coder.profile.test.analyzePath` function. The object stores the identified critical path and the estimated cost percentages of code sections in the critical path.

Use the Code Profile Analyzer to review the results stored in the object.

MaxTime — Maximum time for test generation

positive integer

Maximum time for test generation, specified as a positive integer.

`MaxTime` specifies the maximum time in seconds for test generation. If the specified time is not sufficient for test generation, the function issues an error and suggests increasing the `MaxTime` value.

```
Example: pathCoverageTest =  
coder.profile.test.generateCriticalPathTest(resultsObject,MaxTime=50)
```

Output Arguments

pathCoverageTest — Path of MLDATX test file

character vector

Path of the MLDATX test file, returned as a character vector. The function also returns a path to the MAT data file containing the generated test cases.

Version History

Introduced in R2024a

See Also

`coder.profile.test.analyzePath` | `coder.profile.test.generateTests`

Topics

“Analyze Execution Times of Critical Paths”

coder.profile.test.hiliteCriticalPath

Namespace: coder.profile.test

Highlight critical path in model canvas

Syntax

```
coder.profile.test.hiliteCriticalPath = (criticalPathBlocks)
```

Description

`coder.profile.test.hiliteCriticalPath = (criticalPathBlocks)` highlights the critical path identified by `coder.profile.test.analyzePath` in the model canvas. You can also use Code Profile Analyzer app to highlight the path.

Examples

Highlight Critical Path in Model Canvas

Use `coder.profile.test.hiliteCriticalPath` to highlight the identified critical path in the model canvas programatically.

Open the `CriticalPathIdentify` model.

```
openExample('CriticalPathIdentify');  
model = "CriticalPathIdentify";
```

Identify the critical path by using the `coder.profile.test.analyzePath` function and store the results in `myResults`.

```
myResults = coder.profile.test.analyzePath(model,SimMode="SIL")
```

Get a list of the blocks in the identified path.

```
criticalPathBlocks = coder.profile.test.getCriticalPathElements(myResults)
```

Highlight the identified path in the model canvas.

```
coder.profile.test.hiliteCriticalPath(criticalPathBlocks)
```

Input Arguments

criticalPathBlocks — Blocks in critical path

cell array

Blocks in the critical path, specified as a cell array. Use the `coder.profile.test.getCriticalPathElements` function to get a cell array of blocks in the critical path identified by a critical path analysis.

Version History

Introduced in R2024a

See Also

`coder.profile.test.getCriticalPathElements`

Topics

“Analyze Execution Times of Critical Paths”

coder.RustWrapper.genRustBindings

Create Rust bindings for C code generated from Simulink models

Syntax

```
coder.RustWrapper.genRustBindings(buildInfo)
coder.RustWrapper.genRustBindings(buildInfo,Name=Value)
```

Description

Add-On Required: This feature requires the Simulink Support Package for Rust Code add-on.

`coder.RustWrapper.genRustBindings(buildInfo)` generates Rust® bindings for the C code produced from the Simulink model associated with the specified `buildInfo` object.

`coder.RustWrapper.genRustBindings(buildInfo,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments. For example, set `PackageType` to "lib" to generate a Rust library package, or to "bin" to generate an executable Rust package.

Examples

Generate Rust Library Package in a Custom Folder

Generate a Rust library package for C code generated from the Simulink model `ex_if_else_SL` in a custom folder.

```
open_system("ex_if_else_SL");
slbuild("ex_if_else_SL");
bi=load(fullfile("ex_if_else_SL_ert_rtw", "buildInfo.mat"));
coder.RustWrapper.genRustBindings(bi.buildInfo,PackageType="lib",OutDirPath="C:\rust\bindings");
```

`PackageType` controls the type of Rust package generated. `OutDirPath` controls the folder where the function writes the generated Rust bindings. If you do not specify these name-value arguments, the function uses the build information `bi.buildInfo` to generate Rust bindings for the C code and writes them to the `rust_bindings` folder, which the function creates at the same level as the build folder of the model. For more information on generating Rust bindings using `coder.RustWrapper.genRustBindings`, including prerequisites and setup, see [Generate Rust Bindings for Simulink Generated C Code](#) and [Install Simulink Support Package for Rust Code](#).

Input Arguments

buildInfo — Build information

RTW.`BuildInfo` object

Build information for compiling and linking generated code, specified as an RTW.`BuildInfo` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:

```
coder.RustWrapper.genRustBindings(buildInfo,PackageType="bin",OutDirPath="C:\rust\bindings")
```

PackageType — Type of Rust package to generate

"lib" (default) | "bin"

Specify the type of Rust package to generate. Use "lib" to generate a library package or "bin" to generate a binary package.

Example: `PackageType="lib"`

OutDirPath — Output folder path

character vector | string scalar

Folder where the function writes the generated Rust bindings, specified as a character vector or string scalar. By default, the function writes the bindings to `rust_bindings` folder in the current working folder.

Example: `OutDirPath="C:\rust\bindings"`

Version History

Introduced in R2026a

See Also

`RTW.BuildInfo`

Topics

Generate Rust Bindings for Simulink Generated C Code

"Integrate Rust Code into Simulink"

Simulink Coder Functions

addCompileFlags

Namespace: RTW

Add compiler options to 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 `myBuildInfo` and place the option in the group `OPTS`.

```
myBuildInfo = RTW.BuildInfo;  
addCompileFlags(myBuildInfo,'-O3','OPTS');
```

Add Compiler Flags to OPT_OPTS Group

Add the compiler options `-Zi` and `-Wall` to the build information `myBuildInfo` and place the options in the group `OPT_OPTS`.

```
myBuildInfo = RTW.BuildInfo;  
addCompileFlags(myBuildInfo,'-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 `myBuildInfo`. Place the options `-Zi` and `-Wall` in the group `Debug` and the option `-O3` in the group `MemOpt`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-03'}, ...
    {'Debug' 'MemOpt'});
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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 `'-03'`, the *options* argument is added to the build information as an array of character vectors.

Example: `{'-Zi -Wall' '-03'}`

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' '-03'}` 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 flags in the `'OPTS'`, `'OPT_OPTS'`, and `'OPTIMIZATION_FLAGS'` groups when generating the makefile.

For the build process to consider compiler flags in other groups, the template makefile must contain the token `|>COMPILE_FLAGS_OTHER<|`.

Example: `{'Debug' 'MemOpt'}`

Version History

Introduced in R2006a

See Also

`addDefines` | `addLinkFlags` | `getCompileFlags`

Topics

“Customize Post-Code-Generation Build Processing”

“Customize Template Makefiles”

addDefines

Namespace: RTW

Add preprocessor macro definitions to 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 a macro definition `-DPRODUCTION` to the build information `myBuildInfo` and place the definition in the group `OPTS`.

```
myBuildInfo = RTW.BuildInfo;  
addDefines(myBuildInfo, '-DPRODUCTION', 'OPTS');
```

Add a macro definition `MYDEF` with value `MYVAL`.

```
addDefines(myBuildInfo, '-DMYDEF=MYVAL', 'OPTS');
```

Add a macro definition `MYDEF` with empty value.

```
addDefines(myBuildInfo, '-DMYDEF=', 'OPTS');
```

Add a macro definition `MYDEF` with undefined value.

```
addDefines(myBuildInfo, '-DMYDEF', 'OPTS');
```

Add Macro Definitions to OPT_OPTS Group

Add the macro definitions `-DPROTO` and `-DDEBUG` to the build information `myBuildInfo` and place the definitions in the group `OPT_OPTS`.


```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
    '-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 `myBuildInfo`. Place the definitions `-DPROTO` and `-DDEBUG` in the group `Debug` and the definition `-DPRODUCTION` in the group `Release`.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
    {'-DPROTO -DDEBUG' '-DPRODUCTION'}, ...
    {'Debug' 'Release'});
```

Input Arguments

buildinfo — Name of build information object returned by `RTW.BuildInfo`
object

Object contains information for compiling and linking generated code.

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.

For the build process to consider definitions in other groups, the template makefile must contain the token `|>DEFINES_OTHER<|`.

Example: `{ 'Debug' 'Release' }`

Version History

Introduced in R2006a

See Also

[addCompileFlags](#) | [addLinkFlags](#) | [getDefines](#)

Topics

[“Customize Post-Code-Generation Build Processing”](#)

[“Customize Template Makefiles”](#)

addIncludeFiles

Namespace: RTW

Add include files to 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.

Note The function does **not**:

- Add the file paths to the compiler search path. See `addIncludePaths`.
 - Produce `#include` directives in the generated code
-

Examples

Add Included File to SysFiles Group

Add the include file `mytypes.h` to the build information `myBuildInfo` and place the file in the group `SysFiles`.

```
myBuildInfo = RTW.BuildInfo;  
addIncludeFiles(myBuildInfo, ...  
    '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 `myBuildInfo`, and place the files in the group `AppFiles`.

```
myBuildInfo = RTW.BuildInfo;  
addIncludeFiles(myBuildInfo, ...
```

```
{'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 `myBuildInfo`. Group the files `etc.h` and `etc_private.h` with the character vector `AppFiles` and the file `mytypes.h` with the character vector `SysFiles`.

```
myBuildInfo = RTW.BuildInfo;  
addIncludeFiles(myBuildInfo, ...  
{'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 `myBuildInfo`, and place the files in the group `HFiles`.

```
myBuildInfo = RTW.BuildInfo;  
addIncludeFiles(myBuildInfo, ...  
'*.h', '/proj/src', 'HFiles');
```

Input Arguments

buildinfo — Name of build information object returned by `RTW.BuildInfo`

object

Object provides information for compiling and linking generated code.

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'`

Version History

Introduced in R2006a

See Also

`addIncludePaths | addSourcePaths | findIncludeFiles | getIncludeFiles |
updateFilePathsAndExtensions | updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

addIncludePaths

Namespace: RTW

Add include paths to 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 function adds the file paths to the compiler search path.

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 `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addIncludePaths(myBuildInfo,...  
    '/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 `myBuildInfo` and place the files in the group `etc`.

```
myBuildInfo = RTW.BuildInfo;  
addIncludePaths(myBuildInfo,...  
    {'/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 `myBuildInfo`. 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.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,...
    {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
    '/common/lib'}, {'etc' 'etc' 'shared'});
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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'

Version History

Introduced in R2006a

See Also

addIncludeFiles | addSourceFiles | addSourcePaths | getIncludePaths |
removeIncludePaths | updateFilePathsAndExtensions | updateFileSeparator

Topics

“Customize Post-Code-Generation Build Processing”

addLinkFlags

Namespace: RTW

Add link options to 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 `myBuildInfo` and place the option in the group `OPTS`.

```
myBuildInfo = RTW.BuildInfo;  
addLinkFlags(myBuildInfo,'-T','OPTS');
```

Add Linker Flags to OPT_OPTS Group

Add the linker options -MD and -Gy to the build information `myBuildInfo` and place the options in the group `OPT_OPTS`.

```
myBuildInfo = RTW.BuildInfo;  
addLinkFlags(myBuildInfo,'-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 `myBuildInfo`. Place the options -MD and -Gy in the group `Debug` and the option -T in the group `Temp`.


```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo, {'-MD -Gy' '-T'}, ...
    {'Debug' 'Temp'});
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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' }`

Version History

Introduced in R2006a

See Also

`addCompileFlags` | `addDefines` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing”

addLinkObjects

Namespace: RTW

Add link objects to 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*, *linkobjs*, 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 `myBuildInfo`. Mark both objects as link-only. Since individual priorities are not specified, the function adds the objects to the vector in the order specified.

```
myBuildInfo = RTW.BuildInfo;
addLinkObjects(myBuildInfo,{'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 `myBuildInfo`. 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.

```
myBuildInfo = RTW.BuildInfo;
addLinkObjects(myBuildInfo, {'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 `myBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Mark both objects as precompiled. Group them under the name `MyTest`.

```
myBuildInfo = RTW.BuildInfo;
addLinkObjects(myBuildInfo,{'libobj1' 'libobj2'}, ...
    {'/proj/lib/lib1' '/proj/lib/lib2'},[26 10], ...
    true,false,'MyTest');
```

Input Arguments

buildinfo — Build information object

object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

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'`

Version History

Introduced in R2006a

See Also

`addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `findIncludeFiles` | `getIncludeFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

addNonBuildFiles

Namespace: RTW

Add nonbuild-related files to 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 `myBuildInfo`, and place the file in the group `DocFiles`.

```
myBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myBuildInfo, ...
    '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 `myBuildInfo`, and place the files in the group `DLLFiles`.

```
myBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myBuildInfo, ...
    {'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 `myBuildInfo`, and place the files in the group `DLLFiles`.

```
myBuildInfo = RTW.BuildInfo;  
addNonBuildFiles(myBuildInfo, ...  
    '*.dll', '/proj/dlls', 'DLLFiles');
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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'

Version History

Introduced in R2008a

See Also

getNonBuildFiles

Topics

“Customize Post-Code-Generation Build Processing”

addPreincludeFiles

Namespace: RTW

Add preinclude files to build information

Syntax

`addPreincludeFiles(buildinfo,filenames,paths,groups)`

Description

`addPreincludeFiles(buildinfo,filenames,paths,groups)` adds preinclude files and paths to the build information.

Examples

Add preinclude Files to Build Information

This example shows how to add preinclude filenames to a build information object.

Suppose you have two files, `myFile.c` and `myInc.h`, in your current working folder.

```
// myFile.c
#include <stdio.h>
int main(void) {
    #ifdef __STDC_VERSION__
        printf("__STDC_VERSION__ is defined\n");
    #else
        printf("__STDC_VERSION__ is not defined\n");
    #endif

    #ifdef MY_MACRO
        printf("MY_MACRO is defined\n");
    #else
        printf("MY_MACRO is not defined\n");
    #endif
}
```

```
// myInc.h
#ifndef MYINC_H
#define MYINC_H

#define MY_MACRO

#endif
```

Create an `RTW.BuildInfo` object and specify source files, undefine directives, and preinclude files.

```
buildInfo = RTW.BuildInfo;
buildInfo.ComponentName = 'MyComponent';
```



```
addSourceFiles(buildInfo, 'myfile.c');  
addUndefines(buildInfo, '__STDC_VERSION__');  
addPreincludeFiles(buildInfo, 'myinc.h', pwd);
```

Input Arguments

buildinfo — Build information object

object

Required. RTW.BuildInfo object that contains information for compiling and linking generated code.

filenames — preinclude file list

character vector | array of character vectors | string scalar

Required. The list of preinclude header files for the compiler.

paths — preinclude file paths

character vector | array of character vectors | string scalar

Optional. Specify file paths for the preinclude header files.

groups — File groups

character vector | array of character vectors | string

Optional. Specify groups for the preinclude header files.

Version History

Introduced in R2024a

See Also

[addUndefines](#) | [getUndefines](#) | [removeUndefines](#) | [getPreincludeFiles](#)

Topics

“Define Microsoft Visual Studio Toolchain for Building 32-Bit Application”

addUndefines

Namespace: RTW

Add undefine preprocessor directives to build information

Syntax

`addUndefines(buildinfo,undefs,groups)`

Description

`addUndefines(buildinfo,undefs,groups)` adds undefine preprocessor directives to the build information.

Examples

Add undefine Preprocessor Directives to Build Information

This example shows how to add undefine preprocessor directives to a build information object.

Suppose you have two files, `myFile.c` and `myInc.h`, in your current working folder.

```
// myFile.c
#include <stdio.h>
int main(void) {
    #ifdef __STDC_VERSION__
        printf("__STDC_VERSION__ is defined\n");
    #else
        printf("__STDC_VERSION__ is not defined\n");
    #endif

    #ifdef MY_MACRO
        printf("MY_MACRO is defined\n");
    #else
        printf("MY_MACRO is not defined\n");
    #endif
}
```

```
// myInc.h
#ifndef MYINC_H
#define MYINC_H

#define MY_MACRO

#endif
```

Create an `RTW.BuildInfo` object and specify source files, undefine directives, and preinclude files.

```
buildInfo = RTW.BuildInfo;
buildInfo.ComponentName = 'MyComponent';
```

```
addSourceFiles(buildInfo, 'myfile.c');  
addUndefines(buildInfo, '__STDC_VERSION__');  
addPreincludeFiles(buildInfo, 'myinc.h', pwd);
```

Input Arguments

buildinfo — Build information object

object

Required. RTW.BuildInfo object that contains information for compiling and linking generated code.

undefs — undefine preprocessor directives

character vector | array of character vectors | string

Required. Append set of undefine preprocessor directives to the RTW.BuildInfo object.

groups — Compiler option groups

character vector | array of character vectors | string

Optional. Associate undefine preprocessor directives with groups.

Version History

Introduced in R2024a

See Also

getUndefines | removeUndefines | addPreincludeFiles | getPreincludeFiles

Topics

“Define Microsoft Visual Studio Toolchain for Building 32-Bit Application”

addSourceFiles

Namespace: RTW

Add source files to 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 `myBuildInfo` and place the file in the group `Drivers`.

```
myBuildInfo = RTW.BuildInfo;  
addSourceFiles(myBuildInfo,'driver.c', ...  
    '/proj/src', 'Drivers');
```

Add Source Files to a Group

Add the source files `test1.c` and `test2.c` to the build information `myBuildInfo` and place the files in the group `Tests`.

```
myBuildInfo = RTW.BuildInfo;  
addSourceFiles(myBuildInfo, ...  
    {'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 `myBuildInfo`. Group the files `test1.c` and `test2.c` with the character vector `Tests`. Group the file `driver.c` with the character vector `Drivers`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo, ...
    {'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 myBuildInfo and place the files in the group CFiles.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo, ...
    '*.c', '/proj/src', 'CFiles');
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that holds information for compiling and linking generated code.

Example: myBuildInfo

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'

Version History

Introduced in R2006a

See Also

[addIncludeFiles](#) | [addIncludePaths](#) | [addSourcePaths](#) | [getSourceFiles](#) | [removeSourceFiles](#) | [updateFilePathsAndExtensions](#) | [updateFileSeparator](#)

Topics

“Customize Post-Code-Generation Build Processing”

addSourcePaths

Namespace: RTW

Add source paths to 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 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 `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addSourcePaths(myBuildInfo, ...  
    '/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 `myBuildInfo` and place the files in the group `etc`.

```
myBuildInfo = RTW.BuildInfo;  
addSourcePaths(myBuildInfo, ...  
    {'/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 `myBuildInfo`. Group the paths `/etcproj/etclib` and `/etcproj/etc/etc_build` with the character vector `etc` and the path `/common/lib` with the character vector `shared`.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,...
    {'/etcproj/etclib' '/etcproj/etc/etc_build'...
    '/common/lib'}, {'etc' 'etc' 'shared'});
```

Input Arguments

buildinfo — Build information object

object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

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 `'/etcproj/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'`

Version History

Introduced in R2006a

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `getSourcePaths` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

addTMFTokens

Namespace: RTW

Add template makefile (TMF) tokens to 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 project. 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 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 `myBuildInfo`, and place the token in the group `LINK_INFO`.

```
myBuildInfo = RTW.BuildInfo;
addTMFTokens(myBuildInfo, ...
    '|>CUSTOM_OUTNAME<|', 'foo.exe', 'LINK_INFO');
```

Apply Build Information as Tokens in TMF Build

In the TMF for the target selected for your project, 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 — Build information object
object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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`'

Version History

Introduced in R2009b

See Also

Topics

“Customize Post-Code-Generation Build Processing”

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.

Version History

Introduced in R2018b

See Also

Topics

“Check Code Generation Assumptions”

codebuild

Compile and link generated code

Syntax

```
buildResults = codebuild(buildFolder)
codebuild(buildFolder, Name,Value)
codebuild(buildInfo, Name,Value)
```

Description

`buildResults = codebuild(buildFolder)` loads data from the `buildInfo.mat` file in `buildFolder`, generates a makefile in `buildFolder`, and uses the specified toolchain or template makefile to compile source code that is registered in the `RTW.BuildInfo` object. If the object is at the top of a hierarchy, the function performs the process for each object in the hierarchy.

The function saves compilation artifacts, including object code files, in `buildFolder`.

The function returns an object that contains the display output and the path to the final product (for example, an executable or static library file). To view the display output, run `disp(buildResults)`. To access the display output programmatically, use the contents of the `Log` property. To determine the final product, use the contents of the `FinalProduct` property, e.g., `finalProduct = buildResults.FinalProduct`.

`codebuild(buildFolder, Name,Value)` specifies additional options using one or more name-value pairs.

`codebuild(buildInfo, Name,Value)` compiles the code specified in `buildInfo` using one or more name-value pairs.

Examples

Relocate and Compile Generated Code

For an example that shows how to relocate and compile generated code in another development environment, see [Compile Code in Another Development Environment](#).

Input Arguments

buildFolder — Build folder

character vector | string

Path to the build folder, which typically contains the generated source code. The folder must contain the `buildInfo.mat` file.

Example: `codebuild(pathToCodeFolder, 'BuildMethod', myToolchain)`

buildInfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

Example: `codebuild(myBuildInfo, 'BuildMethod', 'CMake')`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `codebuild(pathToCodeFolder, 'BuildMethod', myToolchain)`

BuildMethod — Build method

character vector | string

Use one of these build methods:

- Toolchain — Specify the name of the toolchain, for example, 'GNU gcc/g++ | gmake (64-bit Linux)'.
- Template makefile — Specify the path to the template makefile.
- CMake — Specify 'cmake', which generates CMakeLists.txt configuration files for the CMake build system. The argument value is case-insensitive. For example, you can also specify 'Cmake' or 'CMake'.

Example: `codebuild(pathToCodeFolder, 'BuildMethod', 'CMake')`

BuildVariant — Build variant

'STANDALONE_EXECUTABLE' | 'MODEL_REFERENCE_CODER' | 'MEX_FILE' | 'SHARED_LIBRARY' | 'STATIC_LIBRARY'

Specify the type of build output:

- 'STANDALONE_EXECUTABLE' -- Generates a standalone, executable file.
- 'MODEL_REFERENCE_CODER' -- Generates a static library.
- 'MEX_FILE' -- Generates a MEX file. Use this value only for building a simulation target, for example, model reference simulation target (ModelReferenceSimTarget) and accelerator mode.
- 'SHARED_LIBRARY' -- Generates a dynamic library.
- 'STATIC_LIBRARY' -- Generates a static library.

Example: `codebuild(pathToCodeFolder, 'BuildVariant', 'SHARED_LIBRARY')`

Output Arguments

buildResults — Build results

object

Capture the display output and the path to the final product (for example, an executable or static library file). To view the display output, in the Command Window, run `disp(buildResults)`. To access the display output programmatically, use the contents of the `Log` property. To determine the final product, use the contents of the `FinalProduct` property. For example:

```
finalProduct = buildResults.FinalProduct
```

Version History

Introduced in R2020b

See Also

`slbuild` | `RTW.BuildInfo`

Topics

Compile Code in Another Development Environment

coder.buildstatus.close

Close Build Status window

Syntax

```
coder.buildstatus.close()  
coder.buildstatus.close(model)  
coder.buildstatus.close(subsystem)
```

Description

`coder.buildstatus.close()` closes Build Status windows.

The Build Status window supports parallel builds of referenced model hierarchies. Do not use the Build Status window for serial builds.

`coder.buildstatus.close(model)` closes the Build Status window for `model`.

`coder.buildstatus.close(subsystem)` closes the Build Status window for `subsystem`.

Examples

Close Build Status Windows

Close Build Status windows that are open.

```
coder.buildstatus.close()
```

Close Build Status Window for a Model

After generating code for `CounterModel`, close the Build Status window for the model.

```
coder.buildstatus.close('CounterModel')
```

Close Build Status Window for a Subsystem

Close the Build Status window for the subsystem 'Amplifier' in model 'CounterModel'.

```
coder.buildstatus.close('CounterModel/Amplifier')
```

Input Arguments

model — Model name

character vector

Model name specified as a character vector

Example: 'CounterModel'

Data Types: char

subsystem — Subsystem name

character vector

Subsystem name specified as a character vector

Example: 'CounterModel/Amplifier'

Data Types: char

Version History

Introduced in R2018a

See Also

`coder.buildstatus.open` | `coder.report.close` | `slbuild`

Topics

"Monitor Parallel Building of Referenced Models"

coder.buildstatus.open

Open Build Status window

Syntax

```
coder.buildstatus.open(model)
coder.buildstatus.open(model,systemTarget)
```

Description

`coder.buildstatus.open(model)` opens the Build Status window for `model`.

The Build Status window supports parallel builds of referenced model hierarchies. Do not use the Build Status window for serial builds.

If the current working folder is the model build folder and the folder contains information from a previous parallel build, opening the Build 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 Status window for `model` and displays the `model` tab. The available tabs are **Simulation Targets** and **Code Generation Targets**

Examples

Open Build Status Window for a Model

After generating code for model 'ParallelBuildA1', open the Build Status window for the model.

```
coder.buildstatus.open('ParallelBuildA1')
```

Open Build Status Window with Simulation Targets

Open the Build Status window for the model 'ParallelBuildA1' and display the **Simulation Targets** tab.

```
coder.buildstatus.open('ParallelBuildA1','sim')
```

Input Arguments

model — Model name

character vector | string scalar

Model name specified as a character vector or a string scalar

Example: 'ParallelBuildA1'

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, the dialog box opens both the **Simulation Targets** tab and **Code Generation Targets** tab. If build information for a target is not available, the dialog box does not open the corresponding system targets tab.

Example: 'rtw'

Data Types: `char` | `string`

Version History

Introduced in R2018a

See Also

`coder.buildstatus.close` | `coder.report.open` | `slbuild`

Topics

“Monitor Parallel Building of Referenced Models”

coder.codedescriptor.CodeDescriptor Class

Namespace: `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.

Creation

`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: `'CustomCodeComments'`

Attributes:

GetAccess public

BuildDir — Build folder

character vector (default)

Path of the build folder where the model is built.

Example: `'C:\Users\Desktop\Work\CustomCodeComments_ert_rtw'`

Attributes:

GetAccess public

Methods

Public Methods

<code>getAllDataInterfaceTypes</code>	Return data interface types
<code>getAllFunctionInterfaceTypes</code>	Return function interface types

getArrayLayout	Return array layout of the generated code
getDataInterfaceTypes	Return data interface types in the generated code
getDataInterfaces	Return information of the specified data interface
getFunctionInterfaceTypes	Return function interface types in the generated code
getFunctionInterfaces	Return information of the specified function interface
getReferencedModelCodeDescriptor	Return coder.codedescriptor.CodeDescriptor object for the specified referenced model
getReferencedModelNames	Return names of referenced models
getServiceFunctionDeclaration	Return declaration of service function interface in generated code
getServiceFunctionPrototype	Return prototype of generated service function interface
getServices	Return service interface object
getSignalsOfInterest	Return description of signals of interest
getTunableParameterInitialValue	Return initial value of tunable parameter in generated code as string

Examples

Create coder.codedescriptor.CodeDescriptor Object

- 1 Open the model.

```
openExample('CustomCodeComments')
```
- 2 Build the model.

```
slbuild('CustomCodeComments')
```
- 3 Create a coder.codedescriptor.CodeDescriptor object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')

modelName = 'CustomCodeComments'
BuildDir = 'C:\Users\Desktop\Work\CustomCodeComments_ert_rtw'
```
- 4 Return a list of all available function interface types.

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)

{'Initialize'}
{'Output'     }
{'Update'     }
{'Terminate'  }
```

Version History

Introduced in R2018a

See Also

getCodeDescriptor | coder.descriptor.DataInterface |
 coder.descriptor.FunctionInterface

Topics

“Get Code Description of Generated Code”
 “Create a Service Interface Configuration”

coder.descriptor.BreakpointDataInterface Class

Namespace: `coder.descriptor`

Superclasses: `coder.descriptor.DataInterface`

Return information about tunable breakpoint set data for a lookup table that has tunable parameters

Description

The `coder.descriptor.BreakpointDataInterface` object describes various properties for breakpoint set data for these Lookup Table blocks that have tunable parameters in the generated code:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Interpolation Using Prelookup
- Direct Lookup Table (n-D)
- Sine
- Cosine

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.BreakpointDataInterface` object for each dimension in the lookup table. The `codeDescObj` object is the `coder.codedescptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

The `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects are created if these conditions are true:

- Lookup table data is tunable.
- One of these conditions is true:
 - Breakpoint set data is tunable.
 - Breakpoint set data is nontunable and the block does not use a `Simulink.LookupTable` object.
 - The block uses a `Simulink.LookupTable` object.

Input Arguments

dataInterfaceName — Name of data interface

Parameters

Specify the Parameters data interface type.

Example: Parameters

Properties

Type — Type of data

`coder.descriptor.types` object

The data type associated with the data such as `integer`, `double`, `matrix`, and its properties.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

SID — Simulink identifier

character vector

The Simulink identifier (SID) is a unique number within the model that Simulink assigns to a block.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

GraphicalName — Name of the tunable parameter for the breakpoints

character vector

The name of the associated tunable parameter for the breakpoints.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

VariantInfo — Variant conditions in the model

`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the data interface.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Implementation — Description of implementation of data

`coder.descriptor.DataImplementation` object

Description of how the data in the generated code is implemented. This property describes characteristics such as data type and size. It also 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.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Timing — Data access rate in run-time environment

`coder.descriptor.TimingInterface` object

The rate at which data is accessed in a run-time environment.

Attributes:

GetAccess	public
SetAccess	public

Unit — Physical unit as attribute on signals

character vector

Specified physical units as attributes on signals at the boundaries of model components.

Attributes:

GetAccess	public
SetAccess	public

Range — Range of output value

`coder.descriptor.Range` object

The range of valid values for the block output signals.

Attributes:

GetAccess	public
SetAccess	public

OperatingPoint — Input value to the lookup table relative to each breakpoint

`coder.descriptor.DataInterface` object

To find the input value in the table, the operating point uses relative breakpoint set data.

Attributes:

GetAccess	public
SetAccess	public

SupportTunableSize — Option to generate code that enables tunability of table size

1 (default) | 0

Option to generate code that enables tunability of the effective size of the table, specified as 0 or 1.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

FixAxisMetadata — Description of breakpoint set data

`coder.descriptor.FixAxisMetadata`

Description of breakpoint set data that is either evenly spaced or non-evenly spaced. The `coder.descriptor.FixAxisMetadata` object is created only if the lookup table data is tunable and the breakpoint set data is not tunable.

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

isBreakpointDataInterface Determine whether object is a coder.descriptor.BreakpointDataInterface object

Examples

Get Breakpoint Dataset Information

This example shows how to retrieve the breakpoint data set information.

Build a model

Open and build a model that contains lookup table blocks.

```
open_system('ASAP2Demo');
slbuild('ASAP2Demo');

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 35.888s

Return properties of Lookup Table parameter

Create a code descriptor object for the model.

```
codeDescObj = coder.getCodeDescriptor('ASAP2Demo');
```

Return properties of the Lookup Table parameter.

```
params = getDataInterfaces(codeDescObj, 'Parameters');
```

The `params` variable is an array of `coder.descriptor.DataInterface` and `coder.descriptor.LookupTableDataInterface` objects. The model `ASAP2Demo` contains three Lookup Table blocks. Only two of them have tunable breakpoint set data. The code generator creates only two corresponding `coder.descriptor.LookupTableDataInterface` objects.

Obtain the details of the `Standard_Axis` block by accessing the sixth location in the array.

```
params(6)
```

```
ans =  
LookupTableDataInterface with properties:  
    Type: [1x1 coder.descriptor.types.Type]  
    SID: 'ASAP2Demo:12'  
    GraphicalName: 'tabledata'  
    VariantInfo: [1x0 coder.descriptor.VariantInfo]  
    Implementation: [1x1 coder.descriptor.DataImplementation]  
    Timing: [1x0 coder.descriptor.TimingInterface]  
    Unit: 'rpm'  
    Range: [1x1 coder.descriptor.Range]  
    SupportTunableSize: 0  
    BreakpointSpecification: 'Explicit values'  
    Output: [1x1 coder.descriptor.DataInterface]  
    Breakpoints: [1x2 coder.descriptor.BreakpointDataInterface Sequence]
```

The `Breakpoints` property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details of the breakpoint set attached to the model Lookup Table block by accessing the first location in the array.

```
params(6).Breakpoints(1)
```

```
ans =  
BreakpointDataInterface with properties:  
    Type: [1x1 coder.descriptor.types.Type]  
    SID: 'ASAP2Demo:12'  
    GraphicalName: 'tabledata'  
    VariantInfo: [1x0 coder.descriptor.VariantInfo]  
    Implementation: [1x1 coder.descriptor.DataImplementation]  
    Timing: [1x0 coder.descriptor.TimingInterface]  
    Unit: ''  
    Range: [1x0 coder.descriptor.Range]  
    OperatingPoint: [1x1 coder.descriptor.DataInterface]  
    SupportTunableSize: 0  
    FixAxisMetadata: [1x0 coder.descriptor.FixAxisMetadata]
```

Version History

Introduced in R2020a

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` |
`coder.descriptor.LookupTableDataInterface` | `coder.descriptor.FixAxisMetadata`

coder.descriptor.DataImplementation Class

Namespace: coder.descriptor

Return information about implementation of data in generated code

Description

An object of the `coder.descriptor.DataImplementation` class describes how the generated code implements a data element. This class has the subclass `coder.descriptor.TypedRegion`.

Examples

Get Data Implementation of Inport

Open the model.

```
openExample('CustomCodeComments')
```

Build the model.

```
slbuild('CustomCodeComments')
```

Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```

Return properties of the Inports data interface in the generated code.

```
dataInterfaces = getDataInterfaces(codeDescObj, 'Inports')
```

`dataInterfaces` is an array of `coder.descriptor.DataInterface` objects. Get the details of the first Inport block of the model by accessing the first location in the array.

```
dataInterfaces(1)
```

```

        Type: [1x1 coder.descriptor.types.Double]
        SID: 'CustomCodeComments:1'
    GraphicalName: 'In1'
        VariantInfo: [0x0 coder.descriptor.VariantInfo]
    Implementation: [1x1 coder.descriptor.StructExpression]
        Timing: [1x1 coder.descriptor.TimingInterface]
```

Return properties of the code representation of the first Inport block.

```
dataInterfaces(1).Implementation
```

```

    StructExpression with properties:
        Type: [1x1 coder.descriptor.types.Type]
        BaseRegion: [1x1 coder.descriptor.TypedRegion]
```

```
ElementIdentifier: 'In1'  
Variant: ''
```

Version History

Introduced in R2025a

See Also

`coder.descriptor.DataInterface` | `coder.descriptor.TypedRegion`

coder.descriptor.DataInterface Class

Namespace: 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. The different types of data interfaces are:

- Root-level inports and outports: An interface between the model and external models or systems, for exchanging data.
- Parameters: Local and global parameters that describe the data for the block, lookup table, and the associated breakpoint set data.
- Data Stores: A repository to store global and shared data that can be written and read.
- Internal data: Internal data structures including DWork vectors, block I/O, and zero-crossings.

Creation

`dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.DataInterface` object. The `codeDescObj` object 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 | SharedLocalDataStores | ExternalParameterObjects | ModelParameters | 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.

Unit — Physical unit as attribute on signals

character vector

Specified physical units as attributes on signals at the boundaries of model components.

Range — Range of output value

`coder.descriptor.Range` object

The range of valid values for the block output signals.

Limitations

A bitfield data structure is generated if you select 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.

For data represented by macros, no `coder.descriptor.DataInterface` object is returned. Examples include data assigned to the following storage classes:

- `Define`
- `ImportedDefine`
- `CompilerFlag`

Examples

Get All Data Interface Types

- 1 Open the model.

```
openExample('CustomCodeComments')
```

- 2 Build the model.

```
slbuild('CustomCodeComments')
```

- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```

- 4 Return a list of all data interface types in the generated code.

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

```
{'Inports'      }
{'Outports'     }
{'Parameters'   }
{'ExternalParameterObjects'}
{'InternalData' }
```

- 5 Return properties of a specified data interface in the generated code.

```
dataInterfaces = getDataInterfaces(codeDescObj, 'Inports')
```

`dataInterfaces` 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.

```
dataInterfaces(1)
```

```
      Type: [1x1 coder.descriptor.types.Double]
      SID: 'CustomCodeComments:1'
  GraphicalName: 'In1'
    VariantInfo: [0x0 coder.descriptor.VariantInfo]
  Implementation: [1x1 coder.descriptor.StructExpression]
        Timing: [1x1 coder.descriptor.TimingInterface]
```

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaceTypes` | `getDataInterfaces`

Topics

“Get Code Description of Generated Code”

coder.descriptor.EvenSpacingMetadata Class

Namespace: coder.descriptor

Superclasses: coder.descriptor.FixAxisMetadata

Return information about evenly spaced breakpoint set data

Description

The coder.descriptor.EvenSpacingMetadata object describes breakpoint set data that is evenly spaced, such as starting point, breakpoint step size, and number of points.

Creation

params = getDataInterfaces(codeDescObj, dataInterfaceName) creates a coder.descriptor.BreakpointDataInterface object for each dimension in the lookup table. The codeDescObj object is the coder.codedescriptor.CodeDescriptor object created for the model by using the getCodeDescriptor function.

The coder.descriptor.BreakpointDataInterface object has property FixAxisMetadata that contains a coder.descriptor.FixAxisMetadata object. The coder.descriptor.FixAxisMetadata contains a coder.descriptor.EvenSpacingMetadata object if the breakpoint set data is evenly spaced.

Input Arguments

dataInterfaceName — Name of data interface

Parameters

Specify the Parameters data interface type.

Example: Parameters

Properties

StartingValue — Starting value in the breakpoint set data

character vector

The first point in evenly spaced breakpoint set data.

Attributes:

GetAccess	public
SetAccess	public

StepValue — Spacing between evenly spaced breakpoints

character vector

The spacing between points in evenly spaced breakpoint set data. This value represents a power of 2 if the IsPow2 returns 1.

Attributes:

GetAccess	public
SetAccess	public

NumPoints — Total number of breakpoint values
character vector

Total number of points in evenly spaced breakpoint set data.

Attributes:

GetAccess	public
SetAccess	public

IsPow2 — Power of 2
1 | 0

Returns 1 if the value in StepValue is a power of 2.

Attributes:

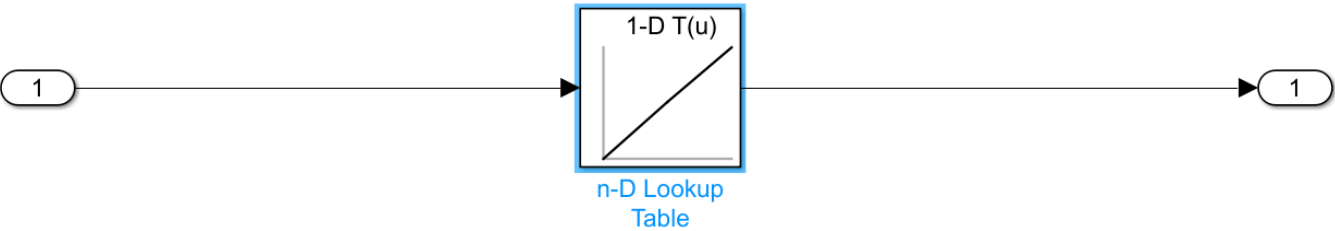
GetAccess	public
SetAccess	public

Data Types: logical

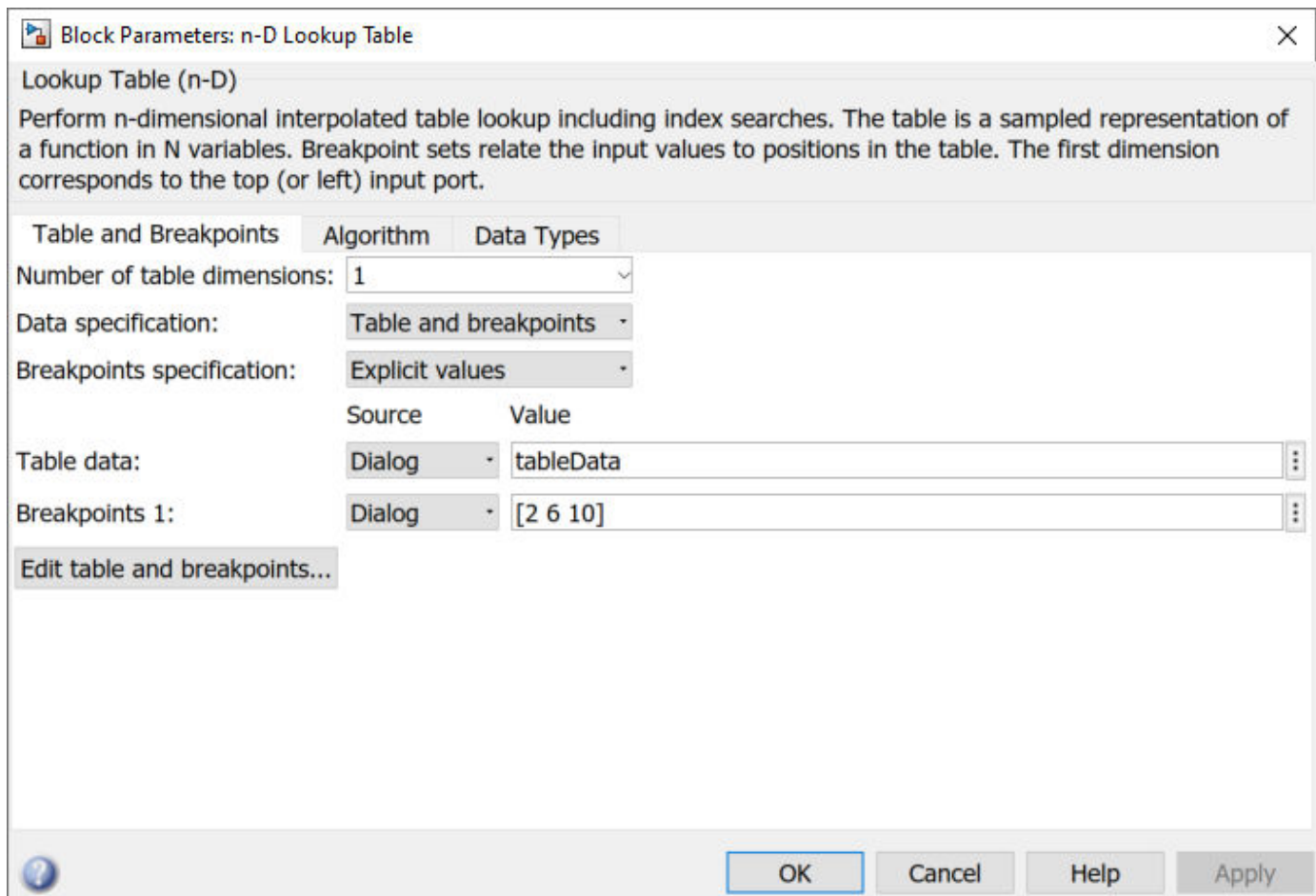
Examples

Get evenly spaced breakpoint set data information

Consider creating model codeDescDemo or a model with similar specifications.



The model contains a n-D Lookup Table. The n-D Lookup Table block takes table data from a model workspace variable named `tableData` with value `[4 5 6]`. The `tableData` is a `Simulink.Parameter` object with non-Auto storage class. The breakpoint set data is specified as `[2 6 10]`.



The model configuration parameter **Default parameter behavior** is set to Inlined.

- 1 Build the model and create a `coder.codeDescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('codeDescDemo')
```

- 2 Retrieve properties of the Lookup Table block and breakpoint set in the generated code.

```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

The `params` variable is an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

```
LookupTableDataInterface with properties:
    Type: [1x1 coder.descriptor.types.Type]
    SID: 'demoModel:22'
    GraphicalName: 'tableData'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x1 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: ''
    Range: [1x1 coder.descriptor.Range]
    SupportTunableSize: 0
    BreakpointSpecification: 'Even spacing'
    Output: [1x1 coder.descriptor.DataInterface]
    Breakpoints: [1x1 coder.descriptor.BreakpointDataInterface Sequence]
```

- 3 The `Breakpoints` property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details

of the breakpoint set attached to the Lookup Table block by accessing the first location in the array.

```
params.Breakpoints(1)
```

BreakpointDataInterface with properties:

```
    Type: [1x1 coder.descriptor.types.Type]
    SID: 'demoModel:22'
    GraphicalName: 'n-D Lookup Table'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x0 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: ''
    Range: [1x1 coder.descriptor.Range]
    OperatingPoint: [1x1 coder.descriptor.DataInterface]
    SupportTunableSize: 0
    FixAxisMetadata: [1x1 coder.descriptor.FixAxisMetadata]
```

- 4 The new `coder.descriptor.FixAxisMetadata` object provides more information about whether the breakpoint set data is evenly spaced or not.

```
params.Breakpoints(1).FixAxisMetadata
```

The information is returned as a new `coder.descriptor.EvenSpacingMetadata` object with these properties:

EvenSpacingMetadata with properties:

```
    StartingValue: 2
    StepValue: 2
    NumPoints: 3
    IsPow2: 1
```

Version History

Introduced in R2020b

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` |
`coder.descriptor.FixAxisMetadata` | `coder.descriptor.BreakpointDataInterface` |
`coder.descriptor.NonEvenSpacingMetadata`

coder.descriptor.FixAxisMetadata Class

Namespace: `coder.descriptor`

Abstract class to get breakpoint set data information

Description

Abstract base class to get breakpoint set data information. Based on the breakpoint set data, you can get either a `coder.descriptor.EvenSpacingMetadata` object or a `coder.descriptor.NonEvenSpacingMetadata` object. To get breakpoint set data information, use the `coder.descriptor.BreakpointDataInterface` object.

You can get a `coder.descriptor.FixAxisMetadata` object if the model meets these conditions:

- Table data is tunable.

Table data is tunable if the Lookup Table block uses a `Simulink.Parameter` object that has a non-Auto storage class or the model configuration parameter **Default parameter behavior** is Tunable.

- Breakpoint set data is not tunable.

Breakpoint set data is tunable if the Lookup Table block uses a `Simulink.Parameter` object that has a non-Auto storage class or the model configuration parameter **Default parameter behavior** is Tunable.

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.BreakpointDataInterface` object for each dimension in the lookup table. The `codeDescObj` object is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

The `coder.descriptor.BreakpointDataInterface` object has property `FixAxisMetadata` that contains a `coder.descriptor.FixAxisMetadata` object.

Input Arguments

dataInterfaceName — Name of data interface

Parameters

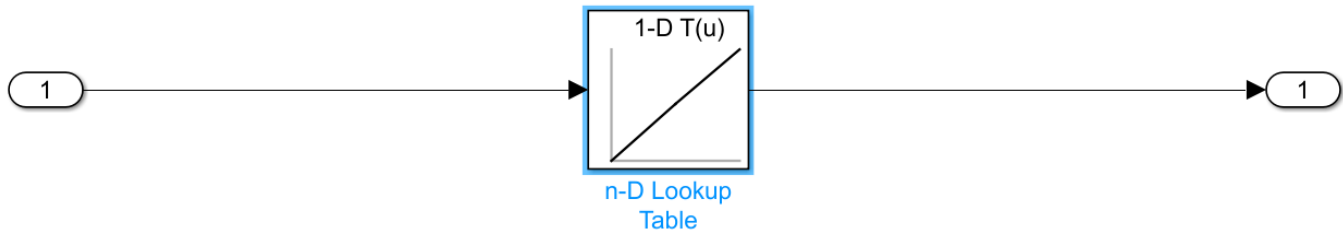
Specify the Parameters data interface type.

Example: Parameters

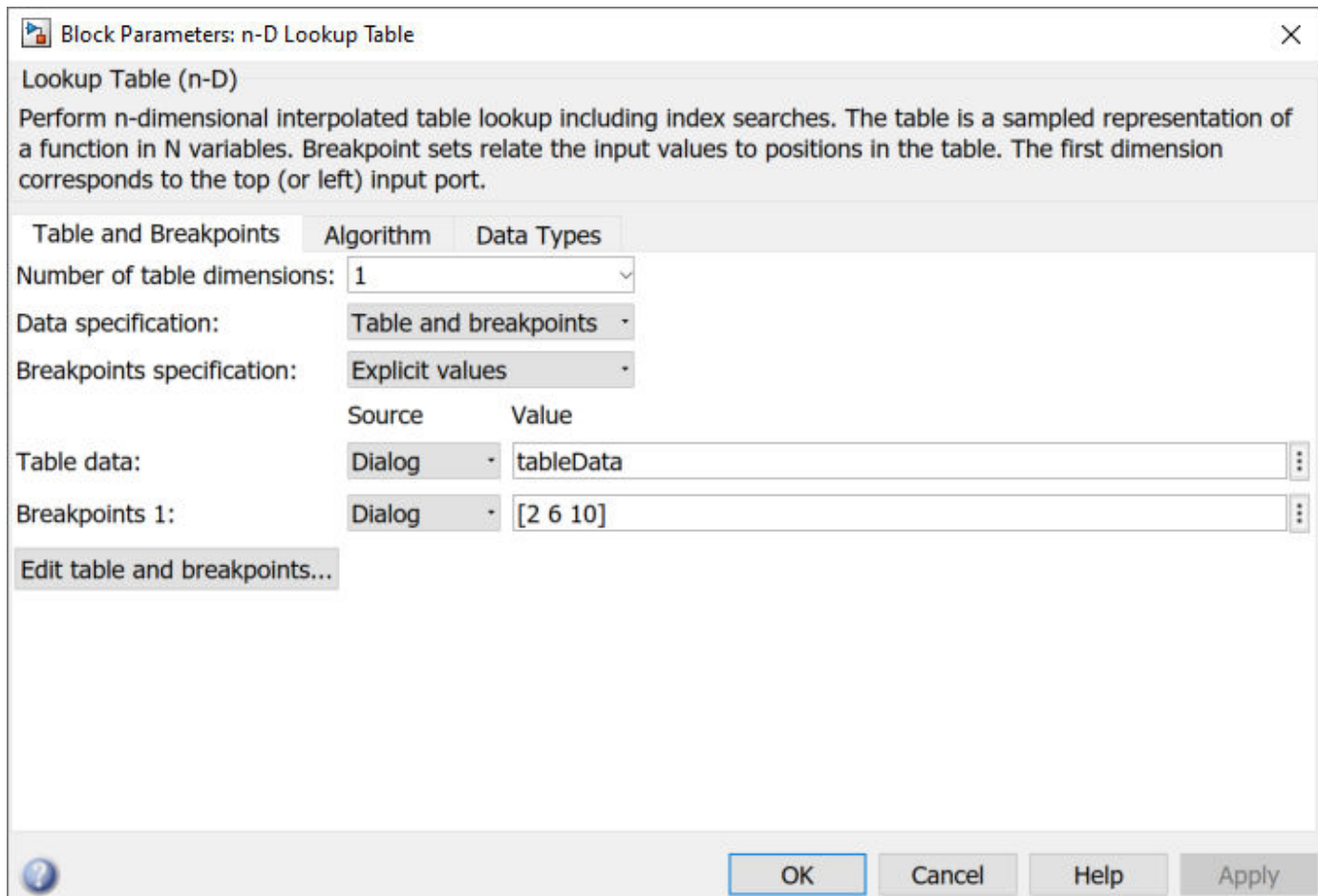
Examples

Get breakpoint set data information

Consider creating model `codeDescDemo` or a model with similar specifications.



The model contains an n-D Lookup Table. The n-D Lookup Table block takes table data from a model workspace variable named `tableData` that has a value of `[4 5 6]`. The `tableData` is a `Simulink.Parameter` object that has a non-Auto storage class. The breakpoint set data is specified as `[2 6 10]`.



The model configuration parameter **Default parameter behavior** is set to Inlined.

- 1 Build the model and create a `coder.codescriptor.CodeDescriptor` object for the model.


```
codeDescObj = coder.getCodeDescriptor('codeDescDemo')
```
- 2 Retrieve properties of the Lookup Table block and breakpoint set in the generated code.


```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

The `params` variable is an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

```
LookupTableDataInterface with properties:
    Type: [1x1 coder.descriptor.types.Type]
    SID: 'demoModel:22'
    GraphicalName: 'tableData'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x1 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: ''
    Range: [1x1 coder.descriptor.Range]
    SupportTunableSize: 0
    BreakpointSpecification: 'Even spacing'
    Output: [1x1 coder.descriptor.DataInterface]
    Breakpoints: [1x1 coder.descriptor.BreakpointDataInterface Sequence]
```

- 3 The `Breakpoints` property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details of the breakpoint set attached to the Lookup Table block by accessing the first location in the array.

```
params.Breakpoints(1)
```

```
BreakpointDataInterface with properties:
    Type: [1x1 coder.descriptor.types.Type]
    SID: 'demoModel:22'
    GraphicalName: 'n-D Lookup Table'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x0 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: ''
    Range: [1x1 coder.descriptor.Range]
    OperatingPoint: [1x1 coder.descriptor.DataInterface]
    SupportTunableSize: 0
    FixAxisMetadata: [1x1 coder.descriptor.FixAxisMetadata]
```

- 4 The new `coder.descriptor.FixAxisMetadata` object provides more information about whether the breakpoint set data is evenly spaced or not.

```
params.Breakpoints(1).FixAxisMetadata
```

The information is returned as a new `coder.descriptor.EvenSpacingMetadata` object that has these properties:

```
EvenSpacingMetadata with properties:
    StartingValue: 2
    StepValue: 2
    NumPoints: 3
    IsPow2: 1
```

- 5 If the breakpoint set data value is changed to `[1 5 10]`, the information is returned as a new `coder.descriptor.NonEvenSpacingMetadata` object that has these properties:

```
NonEvenSpacingMetadata with properties:
    AllPoints: [1x3 Real Sequence]
```

Version History

Introduced in R2020b

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` |
`coder.descriptor.BreakpointDataInterface` | `coder.descriptor.EvenSpacingMetadata`
| `coder.descriptor.NonEvenSpacingMetadata`

coder.descriptor.FunctionInterface Class

Namespace: `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:

- **Allocation:** Contains memory allocation code based on the target of the model. See `model_initialize`.
- **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`.

Creation

`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

`Allocation` | `Initialize` | `Output` | `Update` | `Terminate`

Name of the specified function interface.

Example: `'Output'`

Data Types: `string`

Properties

Prototype — Description of function prototype

`coder.descriptor.Prototype` object

Description of the function prototype, including the function return value, name, arguments, header, and source file, specified as a `coder.descriptor.Prototype` object.

Attributes:

GetAccess	public
SetAccess	public

ActualReturn — Return arguments from function

coder.descriptor.DataInterface object

Return arguments from the function, specified as a coder.descriptor.DataInterface object. When the function returns no data, this field is empty.

Attributes:

GetAccess	public
SetAccess	public

VariantInfo — Variant conditions in model

coder.descriptor.VariantInfo object

Variant conditions in the model that interact with the function interface, specified as a coder.descriptor.VariantInfo object.

Attributes:

GetAccess	public
SetAccess	public

FunctionOwner — Function owner

coder.descriptor.TypedRegion object

Function owner of the class that the function belongs to, specified as a coder.descriptor.TypedRegion object. For C code generation, this property is empty.

Attributes:

GetAccess	public
SetAccess	public

Timing — Function access rate in target environment

coder.descriptor.TimingInterface object

Function access rate in the target environment, specified as a coder.descriptor.TimingInterface object.

Attributes:

GetAccess	public
SetAccess	public

ActualArgs — Input arguments to the function

coder.descriptor.DataInterfaceList object

Input arguments to the function, specified as a coder.descriptor.DataInterfaceList object. When there is no data passed as an argument to the function, this property is empty.

Attributes:

GetAccess public

Examples**Get All Function Interface Types**

- 1 Open the model.
`openExample('CustomCodeComments')`
- 2 Build the model.
`slbuild('CustomCodeComments')`
- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
```

- 4 Return a list of all function interface types in the generated code.

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)

{'Initialize'}
{'Output' }
```

- 5 Return properties of a specified function interface in the generated code.

```
functionInterface = getFunctionInterfaces(codeDescObj, 'Output')

    Prototype: [1x1 coder.descriptor.types.Prototype]
ActualReturn: [0x0 coder.descriptor.DataInterface]
    VariantInfo: [0x0 coder.descriptor.VariantInfo]
        Timing: [1x1 coder.descriptor.TimingInterface]
    ActualArgs: [1x0 coder.descriptor.DataInterface List]
```

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaceTypes` | `getFunctionInterfaces`

Topics

“Get Code Description of Generated Code”

coder.descriptor.LookupTableDataInterface Class

Namespace: coder.descriptor

Superclasses: coder.descriptor.DataInterface

Return information about Lookup Table blocks that have tunable parameters

Description

The `coder.descriptor.LookupTableDataInterface` object describes various properties for these Lookup Table blocks that have tunable parameters in the generated code:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Interpolation Using Prelookup
- Sine
- Cosine

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.LookupTableDataInterface` object if the model has a Lookup Table block that has tunable parameters. The `codeDescObj` object is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

Input Arguments

dataInterfaceName — Name of data interface

Parameters

Specify the Parameters data interface type.

Example: Parameters

Properties

Type — Type of data

`coder.descriptor.types` object

The data type associated with the data such as integer, double, matrix, and its properties.

Attributes:

GetAccess

public

SetAccess

public

SID — Simulink identifier

character vector

The Simulink identifier (SID) is a unique number within the model that Simulink assigns to the block.

Attributes:

GetAccess	public
SetAccess	public

GraphicalName — Name of the tunable parameter for the table data

character vector

The name of the associated tunable parameter for the table data.

Attributes:

GetAccess	public
SetAccess	public

VariantInfo — Variant conditions in the model

`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the data interface.

Attributes:

GetAccess	public
SetAccess	public

Implementation — Description of implementation of data

`coder.descriptor.DataImplementation` object

Description of how the data in the generated code is implemented. This property describes characteristics such as data type and size. It also 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.

Attributes:

GetAccess	public
SetAccess	public

Timing — Data access rate in run-time environment

`coder.descriptor.TimingInterface` object

The rate at which data is accessed in a run-time environment.

Attributes:

GetAccess	public
SetAccess	public

Unit — Physical unit as attribute on signals

character vector

Specified physical units as attributes on signals at the boundaries of model components.

Attributes:

GetAccess	public
SetAccess	public

Range — Range of output value

coder.descriptor.Range object

The range of valid values for the block output signals.

Attributes:

GetAccess	public
SetAccess	public

SupportTunableSize — Tunability of table size

1 (default) | 0

Value that represents whether table is enabled for tunability of the effective size of the table, specified as 0 or 1.

Attributes:

GetAccess	public
SetAccess	public

Data Types: logical

BreakpointSpecification — Source of breakpoint set information in ASAP2 specification

'Explicit values' (default) | 'Reference' | 'Even spacing'

Source of the breakpoint set information, specified as 'Explicit values' (default), 'Even spacing', or 'Reference'. The breakpoint specification is mapped as:

- Explicit values — STD_AXIS
- Reference — COM_AXIS
- Even spacing — FIX_AXIS

For more information on ASAP2 lookup tables, see “Define ASAP2 Information for Lookup Tables”.

Attributes:

GetAccess	public
SetAccess	public

Data Types: char

Output — Data interface for the output of Lookup Table block

coder.descriptor.DataInterface object

Return value of the lookup table operation.

Attributes:

GetAccess	public
SetAccess	public

Breakpoints — Breakpoint set data

`coder.descriptor.BreakpointDataInterface` object

Vector of `coder.descriptor.BreakpointDataInterface` objects that are used in the Lookup Table block. These objects contain the breakpoint set data.

Attributes:

GetAccess public

Methods

Public Methods

`isLookupTableDataInterface` Determine whether object is a `coder.descriptor.LookupTableDataInterface` object

`getAllParameters` Return all associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects

Examples

Get Lookup Table Block Information

This example shows how to retrieve the lookup table block information.

Build a model

Open and build a model that contains lookup table blocks.

```
open_system('ASAP2Demo');
slbuild('ASAP2Demo');

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo

Build Summary

Model reference code generation targets:

Model          Build Reason          Status          Bu
=====
ASAP2DemoModelRef Target (ASAP2DemoModelRef.c) did not exist. Code generated and compiled. 0h

Top model targets:

Model          Build Reason          Status          Bu
=====
ASAP2Demo     Information cache folder or artifacts were missing. Code generated and compiled. 0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 32.643s
```

Return properties of Lookup Table parameter

Create a code descriptor object for the model.

```
codeDescObj = coder.getCodeDescriptor('ASAP2Demo');
```

Return properties of the Lookup Table parameter.

```
params = getDataInterfaces(codeDescObj, 'Parameters');
```

The `params` variable is an array of `coder.descriptor.DataInterface` and `coder.descriptor.LookupTableDataInterface` objects. The model `ASAP2Demo` contains three Lookup Table blocks. Only two of them have tunable breakpoint set data. The code generator creates only two corresponding `coder.descriptor.LookupTableDataInterface` objects.

Obtain the details of the `Standard_Axis` block by accessing the sixth location in the array.

```
params(6)
```

```
ans =
  LookupTableDataInterface with properties:
      Type: [1x1 coder.descriptor.types.Type]
      SID: 'ASAP2Demo:12'
  GraphicalName: 'tabledata'
  VariantInfo: [1x0 coder.descriptor.VariantInfo]
  Implementation: [1x1 coder.descriptor.DataImplementation]
      Timing: [1x0 coder.descriptor.TimingInterface]
      Unit: 'rpm'
      Range: [1x1 coder.descriptor.Range]
  SupportTunableSize: 0
  BreakpointSpecification: 'Explicit values'
      Output: [1x1 coder.descriptor.DataInterface]
  Breakpoints: [1x2 coder.descriptor.BreakpointDataInterface Sequence]
```

Version History

Introduced in R2020a

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` |
`coder.descriptor.BreakpointDataInterface`

coder.descriptor.NonEvenSpacingMetadata Class

Namespace: `coder.descriptor`

Superclasses: `coder.descriptor.FixAxisMetadata`

Return information about non-evenly spaced breakpoint set data

Description

The `coder.descriptor.NonEvenSpacingMetadata` object describes the points in breakpoint set data that are non-evenly spaced.

Creation

`params = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.BreakpointDataInterface` object for each dimension in the lookup table. The `codeDescObj` object is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

The `coder.descriptor.BreakpointDataInterface` object has property `FixAxisMetadata` that contains a `coder.descriptor.FixAxisMetadata` object. The `coder.descriptor.FixAxisMetadata` further contains a `coder.descriptor.NonEvenSpacingMetadata` object if the breakpoint set data is non-evenly spaced.

Input Arguments

dataInterfaceName — Name of data interface

Parameters

Specify the Parameters data interface type.

Example: Parameters

Properties

AllPoints — All values in the breakpoint set

double vector

All values that are in the non-evenly spaced breakpoint set data.

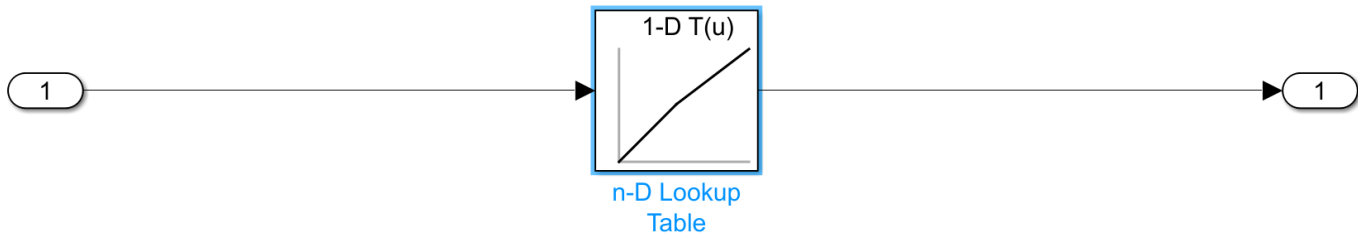
Attributes:

GetAccess public

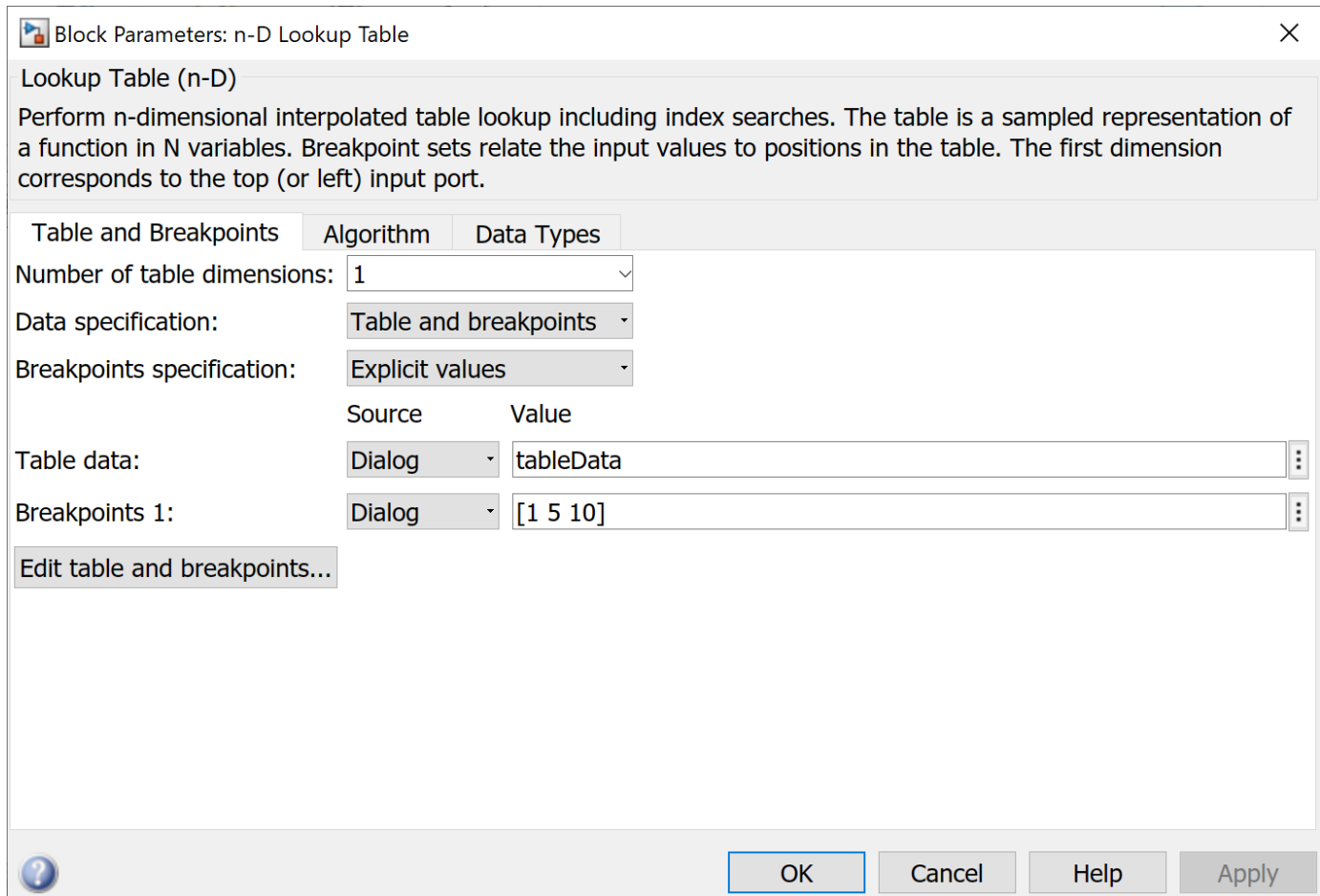
Examples

Get non-evenly spaced breakpoint set data information

Consider creating model `codeDescDemo` or a model with similar specifications.



The model contains an n-D Lookup Table. The n-D Lookup Table block takes table data from a model workspace variable named `tableData` that has a value of `[4 5 6]`. The `tableData` is a `Simulink.Parameter` object with a non-Auto storage class. The breakpoint set data is specified as `[1 5 10]`.



The model configuration parameter **Default parameter behavior** is set to Inlined.

- 1 Build the model and create a `coder.codedescriptor.CodeDescriptor` object for the model.


```
codeDescObj = coder.getCodeDescriptor('codeDescDemo')
```
- 2 Retrieve properties of the Lookup Table block and breakpoint set in the generated code.


```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

The `params` variable is an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

```
LookupTableDataInterface with properties:
    Type: [1x1 coder.descriptor.types.Type]
    SID: 'demoModel:22'
    GraphicalName: 'tableData'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x1 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: ''
    Range: [1x1 coder.descriptor.Range]
    SupportTunableSize: 0
    BreakpointSpecification: 'Even spacing'
    Output: [1x1 coder.descriptor.DataInterface]
    Breakpoints: [1x1 coder.descriptor.BreakpointDataInterface Sequence]
```

- 3 The `Breakpoints` property of the `coder.descriptor.LookupTableDataInterface` object holds a vector of `coder.descriptor.BreakpointDataInterface` objects. Obtain the details of the breakpoint set attached to the Lookup Table block by accessing the first location in the array.

```
params.Breakpoints(1)
```

```
BreakpointDataInterface with properties:
    Type: [1x1 coder.descriptor.types.Type]
    SID: 'demoModel:22'
    GraphicalName: 'n-D Lookup Table'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x0 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: ''
    Range: [1x1 coder.descriptor.Range]
    OperatingPoint: [1x1 coder.descriptor.DataInterface]
    SupportTunableSize: 0
    FixAxisMetadata: [1x1 coder.descriptor.FixAxisMetadata]
```

- 4 The new `coder.descriptor.FixAxisMetadata` object gives you more information about whether the breakpoint set data is evenly spaced or not.

```
params.Breakpoints(1).FixAxisMetadata
```

The information is returned as a new `coder.descriptor.NonEvenSpacingMetadata` object that has these properties:

```
NonEvenSpacingMetadata with properties:
    AllPoints: [1x3 Real Sequence]
```

Version History

Introduced in R2020b

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` |
`coder.descriptor.FixAxisMetadata` | `coder.descriptor.BreakpointDataInterface` |
`coder.descriptor.EvenSpacingMetadata`

coder.descriptor.SignalOfInterest Class

Namespace: `coder.descriptor`

Return information about signals of interest in generated code

Description

An object of the `coder.descriptor.SignalOfInterest` class describes properties of a test point or logged signal in the generated code.

Use the `coder.descriptor.getSignalsOfInterest` method to create a list of `coder.descriptor.SignalOfInterest` objects.

Properties

DataInterface — Data interface of signal

`coder.descriptor.DataInterface` object

Data interface of the signal of interest, specified as a `coder.descriptor.DataInterface` object.

Version History

Introduced in R2025a

See Also

`coder.descriptor.DataInterface` | `getSignalsOfInterest`

coder.descriptor.TypedRegion Class

Namespace: `coder.descriptor`

Superclasses: `coder.descriptor.DataImplementation`

Return information about implementation of typed data in generated code

Description

An object of the `coder.descriptor.TypedRegion` class describes a data element that the generated code implements as a typed data expression. Examples of typed data expressions include variables, or members of a C/C++ structure or array.

Methods

Public Methods

<code>getAddress</code>	Return address expression of data element
<code>getExpression</code>	Return code expression of data element
<code>isDefined</code>	Determine if data element is defined in generated code

Version History

Introduced in R2025a

See Also

`coder.descriptor.DataImplementation`

coder.descriptor.types.Aggregate Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Type`

Return information about variable of aggregate type

Description

An object of the `coder.descriptor.types.Aggregate` class describes a variable in the generated code that is of an aggregate type.

Aggregate types comprise structures and unions. For information about the Code Descriptor classes that represent these types, see `coder.descriptor.types.Struct` and `coder.descriptor.types.Union`.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- 1 — The type qualifier contains `volatile`.
- 0 — The type qualifier does not contain `volatile`.

Data Types: `logical`

Elements — List of elements of aggregate type

vector of `coder.descriptor.types.AggregateElement`

List of objects that represent the elements of aggregate type, specified as a vector of `coder.descriptor.types.AggregateElement` objects. For example, the fields of a `struct` or members of a `union`.

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Struct` |
`coder.descriptor.types.Union` | `coder.descriptor.types.AggregateElement`

`coder.descriptor.types.AggregateElement` Class

Namespace: `coder.descriptor.types`

Return information about element of aggregate type

Description

An object of the `coder.descriptor.types.AggregateElement` class describes information about an element of a variable of an aggregate type in the generated code. Use objects of this class to retrieve information about `struct` fields and `union` members.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Type — Data type of aggregate element

`coder.descriptor.types.Type`

Data type of the aggregate element, specified as a `coder.descriptor.types.Type` object.

Data Types: `coder.descriptor.types.Type`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Aggregate`

`coder.descriptor.types.Argument` Class

Namespace: `coder.descriptor.types`

Return information about argument associated with function or method in generated code

Description

The Code Descriptor programming interface uses the `coder.descriptor.types.Argument` type to describe function parameters and return values in the generated code.

Properties

Name — Identifier used in generated code

character vector

Identifier used for the argument in the generated code, specified as a character vector.

Data Types: `char`

IOType — Input/output type of argument

enumerated value of `coder.descriptor.types.IOType`

Input/output type of the argument, specified as an enumerated value of `coder.descriptor.types.IOType`. Possible return values are:

- `INPUT`
- `OUTPUT`
- `INPUT_OUTPUT`

Data Types: `coder.descriptor.types.Type`

Type — Data type of argument

`coder.descriptor.types.Type`

Data type of the argument, specified as a `coder.descriptor.types.Type` object.

Data Types: `coder.descriptor.types.Type`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Prototype`

coder.descriptor.types.Bool Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Numeric`

Return information about variable of Boolean type

Description

An object of the `coder.descriptor.types.Bool` class describes a variable in the generated code that is of a Boolean type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

WordLength — Word size of data type

positive integer

Word size of a numeric data type, specified as an integer number of bits.

Data Types: `int32`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Double` |
`coder.descriptor.types.Single` | `coder.descriptor.types.Half` |
`coder.descriptor.types.Integer` | `coder.descriptor.types.Fixed`

coder.descriptor.types.Char Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Scalar`

Return information about variable of character type

Description

An object of the `coder.descriptor.types.Char` class describes a variable in the generated code that is of a character type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

WordLength — Word size of data type

positive integer

Word size of a numeric data type, specified as an integer number of bits.

Data Types: `int32`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Scalar` |
`coder.descriptor.types.Numeric`

coder.descriptor.types.Complex Class

Namespace: coder.descriptor.types

Superclasses: coder.descriptor.types.Type

Return information about variable of complex type

Description

An object of the `coder.descriptor.types.Complex` class describes a variable in the generated code that is of a complex type. Generated C code represents complex types as structures.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

true or 1 | false or 0

Indication that the type qualifier contains `const`, specified as a numeric or logical 1 (`true`) or 0 (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- 1 — The type qualifier contains `const`.
- 0 — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

true or 1 | false or 0

Indication that the type qualifier contains `volatile`, specified as a numeric or logical 1 (`true`) or 0 (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- 1 — The type qualifier contains `volatile`.
- 0 — The type qualifier does not contain `volatile`.

Data Types: `logical`

BaseType — Data type of real and imaginary part of complex number

`coder.descriptor.types.Numeric`

Data type of the real and imaginary part of the complex number, specified as an object of `coder.descriptor.types.Numeric` type. Each part has the same numeric type. In C, this is represented as a structure.

Data Types: `coder.descriptor.types.Numeric`

Version History

Introduced in R2024a

R2026a: Data type of BaseType property changed to `coder.descriptor.types.Numeric`

The data type of the BaseType property is now `coder.descriptor.types.Numeric`. Previously, it was `coder.descriptor.types.Type`.

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Numeric`

coder.descriptor.types.Double Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Numeric`

Return information about variable of double-precision type

Description

An object of the `coder.descriptor.types.Double` class describes a variable in the generated code that is of a double-precision type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

WordLength — Word size of data type

positive integer

Word size of a numeric data type, specified as an integer number of bits.

Data Types: `int32`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Bool` |
`coder.descriptor.types.Single` | `coder.descriptor.types.Half` |
`coder.descriptor.types.Integer` | `coder.descriptor.types.Fixed`

coder.descriptor.types.Enum Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Scalar`

Return information about variable of enum type

Description

An object of the `coder.descriptor.types.Enum` class describes a variable in the generated code that is of an enumeration type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

IsCppScoped — Indication that enumeration is scoped

`false` or `0` (default) | `true` or `1`

Indication that the enumeration variable is a C++ scoped enumeration (enum class), specified as a numeric or logical `1` (`true`) or `0` (`false`). The value of this property is `1` if the variable is a scoped enumeration, and `0` otherwise.

Data Types: `logical`

StorageType — Data type of integer values

`coder.descriptor.types.Integer`

Data type used to store the integer values of the enumeration in the generated code, specified as a `coder.descriptor.types.Integer` object.

Data Types: `coder.descriptor.types.Integer`

Strings — Enumeration identifiers

cell array of character vectors

Identifiers representing enumeration variants in the generated code, specified as a cell array of character vectors.

Data Types: `cell`

Values — Enumeration values

integer vector

Integer values for enumeration variants in the generated code, specified as an integer vector.

Data Types: `int32`

Version History

Introduced in R2024b

R2026a: Data type of StorageType property changed to

`coder.descriptor.types.Integer`

The data type of the `StorageType` property is now `coder.descriptor.types.Integer`. Previously, it was `coder.descriptor.types.Type`.

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Scalar` |
`coder.descriptor.types.Char` | `coder.descriptor.types.Integer`

coder.descriptor.types.Fixed Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Numeric`

Return information about variable of fixed-point type

Description

An object of the `coder.descriptor.types.Fixed` class describes a variable in the generated code that is of a fixed-point type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

WordLength — Word size of data type

positive integer

Word size of a numeric data type, specified as an integer number of bits.

Data Types: `int32`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Bias — Bias of scaling

double

Bias of the scaling, specified as a double.

The slope and the bias determine the scaling of a fixed-point number.

Data Types: `double`

DataTypeMode — Data type name and scaling mode

character vector

Data type name and scaling mode, specified as a character vector with one of these values:

- `'Fixed-point: binary point scaling'` - Fixed-point data type and scaling defined by the word length and fraction length
- `'Fixed-point: slope and bias scaling'` - Fixed-point data type and scaling defined by the slope and bias
- `'Fixed-point: unspecified scaling'` - Fixed-point data type with unspecified scaling
- `'Scaled double: binary point scaling'` - Double data type with fixed-point word length and fraction length information retained
- `'Scaled double: slope and bias scaling'` - Double data type with fixed-point slope and bias information retained
- `'Scaled double: unspecified scaling'` - Double data type with unspecified fixed-point scaling
- `'Double'` - Built-in double
- `'Single'` - Built-in single
- `'Half'` - MATLAB half-precision data type
- `'Boolean'` - Built-in boolean

This value is derived from the slope and bias values.

Scaling — Scaling of data type

character vector

Scaling of the data type, represented as a character vector with one of these values:

- 'BinaryPoint' - Scaling for the `numericType` object is defined by the fraction length.
- 'SlopeBias' - Scaling for the `numericType` object is defined by the slope and bias.
- 'Unspecified' - Temporary setting that is only available at `numericType` object creation, and allows for the automatic assignment of a best-precision binary point scaling.

Slope — Slope of scaling

double

Slope of the scaling, specified as a double.

The slope and the bias determine the scaling of a fixed-point number.

Data Types: double

SlopeAdjustmentFactor — Slope adjustment factor of slope and bias scaled number

positive scalar

Slope adjustment factor of a slope and bias scaled number, specified as a positive scalar.

The slope adjustment is equivalent to the fractional slope of a fixed-point number.

Data Types: double

FixedExponent — Fixed exponent of slope and bias scaled number

integer

Fixed exponent of a slope and bias scaled number, specified as an integer.

Data Types: int32

FractionLength — Fraction length

integer

Fraction length, in bits, specified as an integer.

Fraction length can be greater than word length. For more information, see “Binary Point Interpretation” (Fixed-Point Designer).

Data Types: int32

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Bool` |
`coder.descriptor.types.Double` | `coder.descriptor.types.Single` |
`coder.descriptor.types.Half` | `coder.descriptor.types.Integer`

Topics

“Fixed-Point Numbers in Simulink” (Fixed-Point Designer)

coder.descriptor.types.Half Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Numeric`

Return information about variable of half-precision type

Description

An object of the `coder.descriptor.types.Half` class describes a variable in the generated code that is of a half-precision type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

WordLength — Word size of data type

positive integer

Word size of a numeric data type, specified as an integer number of bits.

Data Types: `int32`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Bool` |
`coder.descriptor.types.Double` | `coder.descriptor.types.Single` |
`coder.descriptor.types.Integer` | `coder.descriptor.types.Fixed`

coder.descriptor.types.Integer Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Numeric`

Return information about variable of integer type

Description

An object of the `coder.descriptor.types.Integer` describes a variable in the generated code that is of an integer type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Scalar` |
`coder.descriptor.types.Bool` | `coder.descriptor.types.Double` |
`coder.descriptor.types.Single` | `coder.descriptor.types.Half` |
`coder.descriptor.types.Fixed`

coder.descriptor.types.Matrix Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Type`

Return information about variable of matrix type

Description

An object of the `coder.descriptor.types.Matrix` class describes a variable in the generated code that is of a matrix type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

BaseType — Data type of matrix elements

`coder.descriptor.types.Type`

Data type of the matrix elements, specified as a `coder.descriptor.types.Type` object. Each element has the same type.

Data Types: `coder.descriptor.types.Type`

Dimensions — Dimensions of matrix

integer vector

The dimensions of the matrix, specified as an integer vector. For example, for a 3-by-4 matrix, this property has the value `[3 4]`.

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type`

coder.descriptor.types.Numeric Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Scalar`

Return information about variable of numeric type

Description

An object of the `coder.descriptor.types.Numeric` class describes a variable in the generated code that is of a numeric type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

WordLength — Word size of data type

positive integer

Word size of a numeric data type, specified as an integer number of bits.

Data Types: `int32`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Bool` |
`coder.descriptor.types.Double` | `coder.descriptor.types.Single` |
`coder.descriptor.types.Half` | `coder.descriptor.types.Integer` |
`coder.descriptor.types.Fixed`

coder.descriptor.types.Pointer Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Type`

Return information about variable of pointer type

Description

An object of the `coder.descriptor.types.Pointer` class describes a variable in the generated code that is of a pointer type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

BaseType — Data type that pointer references

`coder.descriptor.types.Type`

Data type that the pointer references, specified as a `coder.descriptor.types.Type` object.

Data Types: `coder.descriptor.types.Type`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Reference` |
`coder.descriptor.types.Void`

`coder.descriptor.types`.Prototype Class

Namespace: `coder.descriptor.types`

Return information about function prototype in generated code

Description

The Code Descriptor programming interface uses the `coder.descriptor.types.Prototype` type to describe a function prototype in the generated code.

Properties

Name — Identifier used in generated code

character vector

Identifier used for the function prototype in the generated code.

Data Types: `char`

Return — Return value of function

`coder.descriptor.types.Argument`

Return value of the function, returned as a `coder.descriptor.Argument` object.

Data Types: `coder.descriptor.types.Argument`

Arguments — Input arguments of function

vector of `coder.descriptor.Argument` objects

Input arguments of the function, returned as a vector of `coder.descriptor.Argument` objects.

Data Types: `coder.descriptor.types.Argument`

HeaderFile — Filename of header file that declares function prototype

character vector

Filename of the header file that declares the function prototype, returned as a character vector.

Data Types: `char`

DefinitionFile — Filename of definition file that defines function prototype

character vector

Filename of the definition file that defines the function prototype, returned as a character vector.

Data Types: `char`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Argument`

coder.descriptor.types.Reference Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Type`

Return information about variable of reference type

Description

An object of the `coder.descriptor.types.Reference` class describes a variable in the generated code that is of a reference type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

BaseType — Data type that reference represents

`coder.descriptor.types.Type`

Data type that the reference represents, specified as a `coder.descriptor.types.Type` object.

Data Types: `coder.descriptor.types.Type`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Pointer`

coder.descriptor.types.Scalar Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Type`

Return information about variable of scalar type

Description

An object of the `coder.descriptor.types.Scalar` class describes a variable in the generated code that is of a scalar type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Char` |
`coder.descriptor.types.Numeric`

coder.descriptor.types.Single Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Numeric`

Return information about variable of single-precision type

Description

An object of the `coder.descriptor.types.Single` class describes a variable in the generated code that is of a single-precision type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

WordLength — Word size of data type

positive integer

Word size of a numeric data type, specified as an integer number of bits.

Data Types: `int32`

Signed — Indication that data type is signed

`true` or `1` | `false` or `0`

Indication that the data type is signed, specified as a numeric or logical `1` (`true`) if signed or `0` (`false`) if unsigned.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Bool` |
`coder.descriptor.types.Double` | `coder.descriptor.types.Half` |
`coder.descriptor.types.Integer` | `coder.descriptor.types.Fixed`

coder.descriptor.types.Struct Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Aggregate`

Return information about variable of `struct` type

Description

An object of the `coder.descriptor.types.Struct` describes a variable in the generated code that is of a `struct` type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains `const`

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains `volatile`

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

Elements — List of struct fields

vector of `coder.descriptor.types.AggregateElement`

List of struct fields, specified as `coder.descriptor.types.AggregateElement` objects.

Data Types: `coder.descriptor.types.AggregateElement`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Aggregate` |
`coder.descriptor.types.AggregateElement` | `coder.descriptor.types.Union`

coder.descriptor.types.Type Class

Namespace: `coder.descriptor.types`

Return information about data type of modeling element or corresponding variable in generated code

Description

An object of `coder.descriptor.types.Type` class describes information about the type of a variable represented in the Simulink model or in the generated code. The `coder.descriptor.types.Type` class has these subclasses for specific data types:

- `coder.descriptor.types.Scalar`
- `coder.descriptor.types.Complex`
- `coder.descriptor.types.Pointer`
- `coder.descriptor.types.Reference`
- `coder.descriptor.types.Void`
- `coder.descriptor.types.Matrix`
- `coder.descriptor.types.Aggregate`

Some of these subclasses have further subclasses. For more information, see each subclass reference page.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Scalar` | `coder.descriptor.types.Complex` |
`coder.descriptor.types.Pointer` | `coder.descriptor.types.Reference` |
`coder.descriptor.types.Void` | `coder.descriptor.types.Matrix` |
`coder.descriptor.types.Aggregate`

coder.descriptor.types.Union Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Aggregate`

Return information about variable of union type

Description

An object of the `coder.descriptor.types.Union` class describes a variable in the generated code that is of a union type.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains const

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains volatile

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `volatile`.
- `0` — The type qualifier does not contain `volatile`.

Data Types: `logical`

Elements — List of union members

vector of `coder.descriptor.types.AggregateElement`

List of union members, specified as a vector of `coder.descriptor.types.AggregateElement` objects.

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Aggregate` |
`coder.descriptor.types.AggregateElement` | `coder.descriptor.types.Struct`

`coder.descriptor.types.Void` Class

Namespace: `coder.descriptor.types`

Superclasses: `coder.descriptor.types.Type`

Represent base type of void pointer

Description

The Code Descriptor programming interface uses this type to represent the data type of pointers to void (`void*`). A `coder.descriptor.types.Pointer` object that represents a void pointer uses a `coder.descriptor.types.Void` object for the value of its `BaseType` property.

The Code Descriptor programming interface does not use this type to represent the return type of a function that returns `void`.

Properties

Identifier — Name of variable in generated code

character vector

Name of the variable in the generated code for the corresponding modeling element, specified as a character vector.

Data Types: `char`

Name — Name of data type represented in Simulink

character vector

Name of the data type that is represented in Simulink for the corresponding modeling element, specified as a character vector.

Data Types: `char`

ReadOnly — Indication that type qualifier contains `const`

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `const`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If you cannot change the value of the variable during execution, the variable is preceded by the `const` keyword as the type qualifier in the generated code.

- `1` — The type qualifier contains `const`.
- `0` — The type qualifier does not contain `const`.

Data Types: `logical`

Volatile — Indication that type qualifier contains `volatile`

`true` or `1` | `false` or `0`

Indication that the type qualifier contains `volatile`, specified as a numeric or logical `1` (`true`) or `0` (`false`). If the compiler cannot apply optimizations to the variable because its value can change due to external sources, the variable is preceded by the `volatile` keyword as the type qualifier in the generated code.

- 1 — The type qualifier contains `volatile`.
- 0 — The type qualifier does not contain `volatile`.

Data Types: `logical`

Version History

Introduced in R2024a

See Also

`coder.descriptor.types.Type` | `coder.descriptor.types.Pointer`

isBreakpointDataInterface

Determine whether object is a `coder.descriptor.BreakpointDataInterface` object

Syntax

```
breakpointTableDataInterface = isBreakpointDataInterface(parameterObj)
```

Description

`breakpointTableDataInterface = isBreakpointDataInterface(parameterObj)` returns a logical value indicating whether the object is a `coder.descriptor.BreakpointDataInterface` object or not.

Input Arguments

parameterObj — `coder.descriptor.BreakpointDataInterface` object

`coder.descriptor.BreakpointDataInterface`

`coder.descriptor.BreakpointDataInterface` object that represents a breakpoint set in the model.

Data Types: `string`

Output Arguments

breakpointTableDataInterface — logical value

1 | 0

Logical value indicating whether the object is a `coder.descriptor.BreakpointDataInterface`.

Data Types: `logical`

Examples

Determine Breakpoint Data Interface Object

This example shows how to determine if an object is `coder.descriptor.BreakpointDataInterface` object.

Build a model

Open and build a model that contains lookup table blocks.

```
open_system('ASAP2Demo');  
slbuild('ASAP2Demo');
```

```
### Searching for referenced models in model 'ASAP2Demo'.  
### Total of 2 models to build.  
### Starting serial code generation build.  
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
```



```
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 57.597s

Return properties of Lookup Table parameter

Create a code descriptor object for the model.

```
codeDescObj = coder.getCodeDescriptor('ASAP2Demo');
```

Return properties of the Lookup Table parameter.

```
params = getDataInterfaces(codeDescObj,'Parameters');
```

The params variable is an array of `coder.descriptor.DataInterface` and `coder.descriptor.LookupTableDataInterface` objects.

Obtain the details of the breakpoint set attached to the model Lookup Table block by accessing the first location in the array.

```
parameterObj = params(6).Breakpoints(1);
```

Determine if the object stored in `parameterObj` variable is a `coder.descriptor.BreakpointDataInterface` object.

```
breakpointDataInterface = isBreakpointDataInterface(parameterObj)
```

```
breakpointDataInterface = logical
    1
```

The code generator returns a logical value of 1 if `parameterObj` is a `coder.descriptor.BreakpointDataInterface` object. Otherwise, the code generator returns a logical value of 0.

Version History

Introduced in R2020a

See Also

`isLookupTableDataInterface` | `coder.descriptor.BreakpointDataInterface`

getAllDataInterfaceTypes

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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

Get All Data Interface Types

Create a `coder.codedescriptor.CodeDescriptor` object for the required model that is built, then list the available data interface types.

- 1 Open the model.

```
openExample('CustomCodeComments')
```
- 2 Build the model.

```
slbuild('CustomCodeComments')
```
- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```
- 4 Return a list of available data interface types.

```
allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)
```

`allDataInterfaceTypes` has these values:

```
{'Inports'           }  
{'Outports'          }  
{'Parameters'        }  
{'GlobalDataStores'  }  
{'SharedLocalDataStores'}  
{'ExternalParameterObjects' }  
{'ModelParameters'   }  
{'InternalData'      }
```

Inports and **Outports** represent root-level inports and outports, not the inports and outports of subsystems.

Parameters includes all parameters in the model. **ExternalParameterObjects** and **ModelParameters** are subsets of **Parameters**. **ModelParameters** represents parameters that you define within a model, such as parameters in the model workspace. **ExternalParameterObjects** represents parameters that you define in the base workspace or in a data dictionary.

GlobalDataStores represents data stores that are defined by a signal object in the base workspace or in a data dictionary. **SharedLocalDataStores** represents Data Store Memory blocks that have the block parameter **Share across model instances** selected.

InternalData represents data elements that are internal to the model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

For more information, see “C Data Code Interface Configuration for Model Interface Elements”.

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` |
`getDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

getAllFunctionInterfaceTypes

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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

Get All Function Interface Types

Create a `coder.codedescriptor.CodeDescriptor` object for the required model which is built, then list the available function interface types.

- 1 Open the `CustomCodeComments` model.

```
openExample('CustomCodeComments')
```
- 2 Build the model.

```
slbuild('CustomCodeComments')
```
- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```
- 4 Return a list of available function interface types.

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
```

`allFunctionInterfaceTypes` has these values:

```
{'Allocation'}  
{'Initialize'}  
{'Output'    }  
{'Update'    }  
{'Terminate' }
```

Version History

Introduced in R2018a

See Also

`coder.codescriptor.CodeDescriptor` | `getFunctionInterfaceTypes` |
`getFunctionInterfaces` | `getCodeDescriptor` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code”

“Configure Generated C Function Interface for Model Entry-Point Functions”

getArrayLayout

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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**.

Examples

Get Array Layout of Model

Create a `coder.codedescriptor.CodeDescriptor` object for the model that is built, then list the array layout of the generated code.

- 1 Open the `CustomCodeComments` model.

```
openExample('CustomCodeComments')
```
- 2 Specify the model configuration parameter **Array layout** as Row-major. Alternatively, in the command window, use these commands:

```
set_param('CustomCodeComments', 'ArrayLayout', 'Row-major');
```
- 3 Build the model.

```
slbuild('CustomCodeComments')
```
- 4 Create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
5 Return the array layout of the generated code.
arrayLayout = getArrayLayout(codeDescObj)
arrayLayout has this value:
    'Row-major'
```

Version History

Introduced in R2018b

See Also

`coder.codedescriptor.CodeDescriptor` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

“Code Generation of Matrices and Arrays”

getDataInterfaces

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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` | `SharedLocalDataStores` | `ExternalParameterObjects` | `ModelParameters` | `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

Get Data Interfaces of Inports Type

- 1 Open the CustomCodeComments model.

```
openExample('CustomCodeComments')
```

- 2 Build the model.

```
slbuild('CustomCodeComments')
```

- 3 Create a coder.codeDescriptor.CodeDescriptor object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```

- 4 Return a list of all data interface types in the generated code.

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

dataInterfaceTypes has these values:

```
{'Inports'      }  
{'Outports'    }  
{'Parameters'  }  
{'ExternalParameterObjects'}  
{'InternalData'}
```

- 5 Return properties of Inport blocks in the generated code.

```
dataInterfaces = getDataInterfaces(codeDescObj, 'Inports')
```

dataInterfaces 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.

```
dataInterfaces(1)
```

The first coder.descriptor.DataInterface object with properties is returned.

```
      Type: [1x1 coder.descriptor.types.Type]  
      SID: 'CustomCodeComments:99'  
  GraphicalName: 'In1'  
    VariantInfo: [1x0 coder.descriptor.VariantInfo]  
Implementation: [1x1 coder.descriptor.DataImplementation]  
      Timing: [1x1 coder.descriptor.TimingInterface]  
      Unit: ''  
      Range: [1x0 coder.descriptor.Range]
```

Version History

Introduced in R2018a

See Also

coder.codeDescriptor.CodeDescriptor | getAllDataInterfaceTypes |
getDataInterfaceTypes | coder.descriptor.DataInterface

Topics

“Get Code Description of Generated Code”

getDataInterfaceTypes

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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

Get Data Interface Types in Generated Code

- 1 Build the model.

```
openExample('CounterModel.slx')
slbuild('CounterModel')
```
- 2 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CounterModel')
```
- 3 Return a list of data interface types in the generated code.

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

`dataInterfaceTypes` has these values for model `CounterModel`:

```
{'Inports'      }  
{'Outports'    }  
{'InternalData' }
```

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` |
`getDataInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

getFunctionInterfaces

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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

`Allocation` | `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

Get Function Interface of Output Type

- 1 Open the model.

```
openExample('CustomCodeComments')
```

- 2 Build the model.

```
slbuild('CustomCodeComments')
```

- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```

- 4 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 `CustomCodeComments`:

```
{'Initialize'}  
{'Output' }
```

- 5 Return properties of a specified function interface in the generated code.

```
functionInterface = getFunctionInterfaces(codeDescObj, 'Output')
```

`functionInterface` is a `coder.descriptor.FunctionInterface` object.

```
Prototype: [1x1 coder.descriptor.types.Prototype]  
ActualReturn: [0x0 coder.descriptor.DataInterface]  
VariantInfo: [0x0 coder.descriptor.VariantInfo]  
Timing: [1x1 coder.descriptor.TimingInterface]  
ActualArgs: [1x0 coder.descriptor.DataInterface List]
```

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` |
`getFunctionInterfaceTypes` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code”

getFunctionInterfaceTypes

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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

Get Function Interface Types in Generated Code

- 1 Build the model.

```
openExample('CounterModel.slx')
slbuild('CounterModel')
```
- 2 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CounterModel')
```
- 3 Return a list of function interface types in the generated code.

```
functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
```

`functionInterfaceTypes` has these values for model `CounterModel`:

```
{'Initialize'}  
{'Output' }
```

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

getReferencedModelCodeDescriptor

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `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

Get Code Descriptor Object for Referenced Model

Open the model `AsynchronousEventsTop`.

```
openExample('AsynchronousEventsTop');
```

Build the model.

```
slbuild('AsynchronousEventsTop');
```

Create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('AsynchronousEventsTop');
```

Get a list of referenced models included in the model.

```
refModels = getReferencedModelNames(codeDescObj);  
refmodels
```

```
refModels = 1x1 cell array  
    {'AsynchronousEventsRef'}
```

Get the `coder.codedescriptor.CodeDescriptor` object for the referenced model `AsynchronousEventsRef`.

```
refCodeDescriptorObj = getReferencedModelCodeDescriptor(codeDescObj, 'AsynchronousEventsRef');  
refCodeDescriptorObj
```

```
refCodeDescriptorObj =  
    CodeDescriptor with properties:  
        ModelName: 'AsynchronousEventsRef'  
        BuildDir: 'C:\Users\Desktop\Work\slprj\tornado\AsynchronousEventsRef'
```

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelNames` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

getReferencedModelNames

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `coder.codedescriptor`

Return names of referenced models

Syntax

```
refModels = getReferencedModelNames(codeDescObj)
```

Description

`refModels = getReferencedModelNames(codeDescObj)` returns a list of referenced models for a `coder.codedescriptor.CodeDescriptor` object. By default, the list includes only the unprotected models that the `CodeDescriptor` object directly references. For example, if `model1` references `model2` and `model2` references `model3`, then running `getReferencedModelNames` from the `model1` `CodeDescriptor` object returns only `model2`, not `model3`. To retrieve different lists of referenced models, use the optional **IncludeAllLevels** and **IncludeProtectedModels** parameters.

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.

IncludeAllLevels — Whether to return the full referenced model hierarchy

`true` | `false`

Whether to return the full referenced model hierarchy, specified as `true` or `false`. If you set this parameter to `true`, the method recursively searches referenced models and returns the full referenced model hierarchy. If you set this parameter to `false` or do not specify a value, the method returns only the models that the top model directly references.

IncludeProtectedModels — Whether to include protected models

`true` | `false`

Whether to include protected models, specified as `true` or `false`. If you set this parameter to `true`, the method includes protected models in the list of referenced models. If you set this parameter to `false` or do not specify a value, the method excludes protected models from the list.

Output Arguments

refModels — Names of referenced models

cell array of character vectors

A list of referenced models.

Examples

Get Names of Directly-Referenced Models Included in a Model

Open the model `AsynchronousEventsTop`.

```
openExample('AsynchronousEventsTop');
```

Build the model.

```
slbuild('AsynchronousEventsTop');
```

Create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('AsynchronousEventsTop');
```

Get a list of referenced models included in the model.

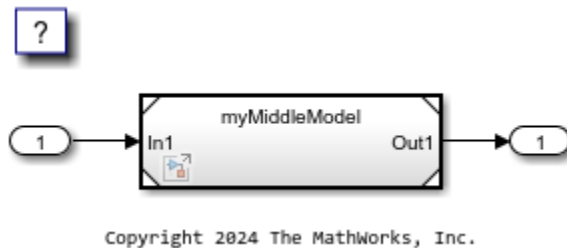
```
getReferencedModelNames(codeDescObj)
```

ans =

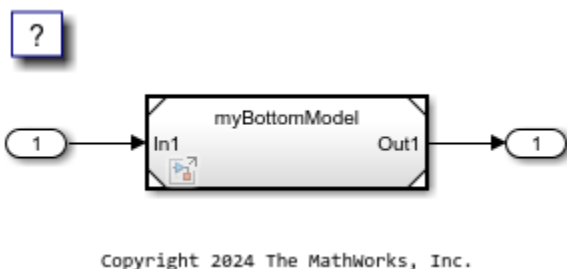
```
1x1 cell array  
{'AsynchronousEventsRef'}
```

Get Names of Nested Referenced Models

Consider this model, `myTopModel`, that references a model, `myMiddleModel`.



`myMiddleModel` contains a reference to another model, `myBottomModel`.



If you build `myTopModel` and use `getReferencedModelNames` without specifying the **IncludeAllLevels** parameter, the output includes only `myMiddleModel`.

```
model = "myTopModel";
slbuild(model);
codeDescObj = coder.getCodeDescriptor(model);
getReferencedModelNames(codeDescObj)
```

ans =

1x1 cell array

```
{'myMiddleModel'}
```

If you set the **IncludeAllLevels** parameter to `true`, the output also includes `myBottomModel`.

```
getReferencedModelNames(codeDescObj, IncludeAllLevels = true)
```

ans =

2x1 cell array

```
{'myMiddleModel'}
{'myBottomModel'}
```

Get Names of Protected Referenced Models

Consider this model, `myTopModel`, that references a protected model, `myProtectedModel`.



Copyright 2024 The MathWorks, Inc.

If you build the model and use `getReferencedModelNames` without specifying the **IncludeProtectedModels** parameter, the output is empty.

```
model = "myTopModel";
slbuild(model);
codeDescObj = coder.getCodeDescriptor(model);
getReferencedModelNames(codeDescObj)
```

ans =

0x1 empty cell array

If you set the **IncludeProtectedModels** parameter to `true`, the output includes `myProtectedModel`.

```
getReferencedModelNames(codeDescObj, IncludeProtectedModels = true)
```

ans =

1x1 cell array

```
{'myProtectedModel'}
```

Version History

Introduced in R2018a

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelCodeDescriptor`

Topics

“Get Code Description of Generated Code”

getSignalsOfInterest

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `coder.codedescriptor`

Return description of signals of interest

Syntax

```
signals = getSignalsOfInterest(codeDescObj,selector)
signals = getSignalsOfInterest(codeDescObj,selector1,selector2,...)
```

Description

`signals = getSignalsOfInterest(codeDescObj,selector)` returns a list of `SignalOfInterest` objects describing signals that match the given selector `selector`.

`signals = getSignalsOfInterest(codeDescObj,selector1,selector2,...)` returns a list of `SignalOfInterest` objects describing signals that match any of the given selectors.

Input Arguments

codeDescObj — Code Descriptor object

`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about the generated code.

Data Types: `coder.codedescriptor.CodeDescriptor`

selector — Selector for signal

'TestPoint' | 'DataLogging'

Selector matching the type of signal for which you want to retrieve information in the generated code, specified as a character vector.

- 'TestPoint' — Return a description of test points in the generated code.
- 'DataLogging' — Return a description of logged signals in the generated code.

Data Types: `char`

Output Arguments

signals — List of signals of interest

vector of `coder.descriptor.SignalOfInterest` objects

List of signals of interest in the generated code, returned as a vector of `coder.descriptor.SignalOfInterest` objects.

Examples

Get Metadata About Code Representation of Test Point

- 1 Open the model.

```
openExample('CustomCodeComments')
```
- 2 Create a test point for the output of the Stateflow® block.

```
portHandles = get_param('CustomCodeComments/Stateflow','portHandles');  
outportHandle = portHandles.Outport;  
set_param(outportHandle,'TestPoint','on')
```
- 3 Build the model.

```
slbuild('CustomCodeComments')
```
- 4 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```
- 5 Retrieve information about the code representation of the test point by calling `getSignalsOfInterest` with the appropriate selector.

```
signals = codeDescObj.getSignalsOfInterest('TestPoint')  
  
signals =  
  
SignalOfInterest with properties:  
    DataInterface: [1x1 coder.descriptor.DataInterface]
```

Version History

Introduced in R2025a

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.SignalOfInterest` | `coder.descriptor.DataInterface`

Topics

“Configure Signals as Test Points”

“Get Metadata About Test Points and Logged Signals in Generated Code”

getTunableParameterInitialValue

Class: `coder.codedescriptor.CodeDescriptor`

Namespace: `coder.codedescriptor`

Return initial value of tunable parameter in generated code as string

Syntax

```
initialCodeValueStr = getTunableParameterInitialValue(codeDescObj,  
parameterDataInterface)
```

Description

`initialCodeValueStr = getTunableParameterInitialValue(codeDescObj, parameterDataInterface)` returns the initial value of a parameter described by the data interface object `parameterDataInterface`.

Input Arguments

codeDescObj — Code Descriptor object

`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for which you want to retrieve the information about the generated code.

Data Types: `coder.codedescriptor.CodeDescriptor`

parameterDataInterface — Parameter data interface

`coder.codedescriptor.DataInterface` object

`coder.codedescriptor.DataInterface` object representing a tunable parameter for which you want to retrieve initial values in the generated code. You can create these objects by calling `getDataInterfaces('Parameters')` on `codeDescObj`.

Data Types: `coder.codedescriptor.DataInterface`

Output Arguments

initialCodeValueStr — Parameter initial value

character vector

Initial value of the parameter described by `parameterDataInterface`, returned as a character vector. The format of the returned value matches that of the generated code.

Data Types: `char`

Limitations

The following parameters are unsupported:

- Images
- Structured parameters, such as arrays and buses, that contain nonfinite values

Examples

Get Initial Parameter Values in Generated Code

- 1 Open the model.
`openExample('CustomCodeComments')`
- 2 Configure block parameters to be tunable in the generated code.
`set_param('CustomCodeComments','DefaultParameterBehavior','Tunable')`
- 3 Build the model.
`slbuild('CustomCodeComments')`
- 4 Create a `coder.codeDescriptor.CodeDescriptor` object for the required model.
`codeDescObj = coder.getCodeDescriptor('CustomCodeComments')`
- 5 Get parameter data interfaces in the generated code.
`dataInterfaces = getDataInterfaces(codeDescObj,'Parameters');`
- 6 Return the properties of the second data interface object, describing parameters for the Constant block in the generated code.
`dataInterface = dataInterfaces(2)`
`ans =`

```
DataInterface with properties:
    Type: [1x1 coder.descriptor.types.Type]
    SID: ''
    GraphicalName: 'const_val'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x1 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: ''
    Range: [1x0 coder.descriptor.Range]
```
- 7 Return the initial parameter values for the Constant block.
`codeDescObj.getTunableParameterInitialValue(dataInterface)`
`ans =`
`{ 1.0, 2.0, 3.0, 4.0 }`

Version History

Introduced in R2026a

See Also

`coder.codeDescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | Default parameter behavior

Topics

“Get Code Description of Generated Code”

“Get Generated Type and Name of Inport Structures”

getAllParameters

Return all associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects

Syntax

```
dataInterface = getAllParameters(parameterObj)
```

Description

`dataInterface = getAllParameters(parameterObj)` returns all associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

Input Arguments

parameterObj — `coder.descriptor.LookupTableDataInterface` object
`coder.descriptor.LookupTableDataInterface` object

The `coder.descriptor.LookupTableDataInterface` object that represents a Lookup Table block in the model.

Data Types: `string`

Output Arguments

dataInterface — array of `coder.descriptor.LookupTableDataInterface` and/or `coder.descriptor.BreakpointDataInterface` objects
`coder.descriptor.LookupTableDataInterface` object | array of `coder.descriptor.LookupTableDataInterface` and/or `coder.descriptor.BreakpointDataInterface` objects

The `coder.descriptor.LookupTableDataInterface` object represents a Lookup Table block in the model. The `coder.descriptor.BreakpointDataInterface` object represents the breakpoint set data associated with the Lookup Table block.

Examples

Get All Lookup Table Data Interface and Breakpoint Data Interface Objects

This example shows how to retrieve all the lookup table data interface and breakpoint data interface objects from a model.

Build a model

Open and build a model that contains lookup table blocks.

```
open_system('ASAP2Demo');  
slbuild('ASAP2Demo');
```

```

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo

```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)

Build duration: 0h 0m 52.535s

Create a code descriptor object for the model.

```
codeDescObj = coder.getCodeDescriptor('ASAP2Demo');
```

Return properties of the Lookup Table parameter.

```
params = getDataInterfaces(codeDescObj, 'Parameters');
```

The params variable is an array of `coder.descriptor.DataInterface` and `coder.descriptor.LookupTableDataInterface` objects.

Obtain the details of the model Lookup Table block by accessing the sixth location in the array.

```
parameterObj = params(6)
```

```
parameterObj =
```

```
LookupTableDataInterface with properties:
```

```

    Type: [1x1 coder.descriptor.types.Type]
    SID: 'ASAP2Demo:12'
    GraphicalName: 'tabledata'
    VariantInfo: [1x0 coder.descriptor.VariantInfo]
    Implementation: [1x1 coder.descriptor.DataImplementation]
    Timing: [1x0 coder.descriptor.TimingInterface]
    Unit: 'rpm'
    Range: [1x1 coder.descriptor.Range]
    SupportTunableSize: 0
    BreakpointSpecification: 'Explicit values'
    Output: [1x1 coder.descriptor.DataInterface]
    Breakpoints: [1x2 coder.descriptor.BreakpointDataInterface Sequence]

```

Retrieve all the associated `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

```
dataInterface = getAllParameters(parameterObj)
```

```
dataInterface =  
    1×3 heterogeneous DataInterface (LookupTableDataInterface, BreakpointDataInterface) array with  
  
    Type  
    SID  
    GraphicalName  
    VariantInfo  
    Implementation  
    Timing  
    Unit  
    Range
```

Version History

Introduced in R2020a

See Also

`coder.descriptor.LookupTableDataInterface` |
`coder.descriptor.BreakpointDataInterface` | `coder.descriptor.DataInterface`

isLookupTableDataInterface

Determine whether object is a `coder.descriptor.LookupTableDataInterface` object

Syntax

```
lookupTableDataInterface = isLookupTableDataInterface(parameterObj)
```

Description

`lookupTableDataInterface = isLookupTableDataInterface(parameterObj)` returns a logical value indicating whether the object is a `coder.descriptor.LookupTableDataInterface` object.

Input Arguments

parameterObj — `coder.descriptor.LookupTableDataInterface` object

`coder.descriptor.LookupTableDataInterface` object

`coder.descriptor.LookupTableDataInterface` object that represents a Lookup Table block in the model.

Data Types: string

Output Arguments

lookupTableDataInterface — logical value

1 | 0

Logical value indicating whether the object is a `coder.descriptor.LookupTableDataInterface`.

Data Types: logical

Examples

Determine Lookup Table Data Interface Object

This example shows how to determine if an object is `coder.descriptor.LookupTableDataInterface` object.

Build a model

Open and build a model that contains lookup table blocks.

```
open_system('ASAP2Demo');
slbuild('ASAP2Demo');
```

```
### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
```

```
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 24.533s

Return properties of Lookup Table parameter

Create a code descriptor object for the model.

```
codeDescObj = coder.getCodeDescriptor('ASAP2Demo');
```

Return properties of the Lookup Table parameter.

```
params = getDataInterfaces(codeDescObj, 'Parameters');
```

The `params` variable is an array of `coder.descriptor.DataInterface` and `coder.descriptor.LookupTableDataInterface` objects. The model `ASAP2Demo` contains three Lookup Table blocks. Only two of them have tunable breakpoint set data. The code generator creates only two corresponding `coder.descriptor.LookupTableDataInterface` objects.

Obtain the details of the `Standard_Axis` block by accessing the sixth location in the array.

```
parameterObj = params(6);
```

Determine if the object stored in `parameterObj` variable is a `coder.descriptor.LookupTableDataInterface` object.

```
lookupTableDataInterface = isLookupTableDataInterface(parameterObj)
```

```
lookupTableDataInterface = logical
    1
```

The code generator returns a logical value of 1 if `parameterObj` is a `coder.descriptor.LookupTableDataInterface` object. Otherwise, the code generator returns a logical value of 0.

Version History

Introduced in R2020a

See Also

`coder.descriptor.LookupTableDataInterface` | `isBreakpointDataInterface`

getAddress

Class: `coder.descriptor.TypedRegion`

Namespace: `coder.descriptor`

Return address expression of data element

Syntax

```
addr = getAddress(implementation,noThrow)
```

Description

`addr = getAddress(implementation,noThrow)` returns a C/C++ expression that enables you to access the address of the data element in the generated code.

Input Arguments

implementation — Description of implementation of data

`coder.descriptor.TypedRegion` object

Description of implementation of data in the generated code, specified as a `coder.descriptor.TypedRegion` object.

noThrow — Flag to throw error if expression is not usable

`false` or `0` (default) | `true` or `1`

Flag to throw an error if the expression is not usable in arbitrary code. The elements of the expression are usable if all members of the `TypedRegion` object exist in the generated code and are not private or protected class members.

- `false` or `0` — Throw an error.
- `true` or `1` — Do not throw an error.

Data Types: `logical`

Output Arguments

addr — C/C++ expression to access address of data element

character vector

C/C++ expression to access the address of the data element in the generated code, relative to the topmost element in the `TypedRegion` object, returned as a character vector.

Data Types: `char`

Examples

Get C Expression to Access Address of Inport Block in Generated Code

- 1 Open the model.

```
openExample('CustomCodeComments')
```
- 2 Build the model.

```
slbuild('CustomCodeComments')
```
- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```
- 4 Return properties of the Inports data interface type in the generated code.

```
dataInterfaces = getDataInterfaces(codeDescObj, 'Inports')
```
- 5 Return properties of the code representation of the first Inport block.

```
implementation = dataInterfaces(1).Implementation
```
- 6 Return the C expression for accessing the address of the first Inport block in the generated code.

```
getAddress(implementation)

ans =
    '&(rtU.In1)'
```

Version History

Introduced in R2025a

See Also

`coder.descriptor.TypedRegion` | `getExpression` | `isDefined`

getExpression

Class: `coder.descriptor.TypedRegion`

Namespace: `coder.descriptor`

Return code expression of data element

Syntax

```
expr = getExpression(implementation,noThrow)
```

Description

`expr = getExpression(implementation,noThrow)` returns a C/C++ expression that enables you to access the data element in the generated code.

Input Arguments

implementation — Description of implementation of data

`coder.descriptor.TypedRegion` object

Description of implementation of data in the generated code, specified as a `coder.descriptor.TypedRegion` object.

noThrow — Flag to throw error if expression is not usable

`false` or `0` (default) | `true` or `1`

Flag to throw an error if the expression is not usable in arbitrary code. The elements of the expression are usable if all members of the `TypedRegion` object exist in the generated code and are not private or protected class members.

- `false` or `0` — Throw an error.
- `true` or `1` — Do not throw an error.

Data Types: `logical`

Output Arguments

expr — C/C++ expression to access data element

character vector

C/C++ expression to access the data element, relative to the topmost element in the `TypedRegion` object, returned as a character vector.

Data Types: `char`

Examples

Get C Expression to Access Inport Block in Generated Code

- 1 Open the model.

```
openExample('CustomCodeComments')
```
- 2 Build the model.

```
slbuild('CustomCodeComments')
```
- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```
- 4 Return properties of the Inports data interface type in the generated code.

```
dataInterfaces = getDataInterfaces(codeDescObj, 'Inports')
```
- 5 Return properties of the code representation of the first Inport block.

```
implementation = dataInterfaces(1).Implementation
```
- 6 Return the C expression for accessing the first Inport block in the generated code.

```
getExpression(implementation)
```

```
ans =  
    'rtU.In1'
```

Version History

Introduced in R2025a

See Also

`coder.descriptor.TypedRegion` | `getAddress` | `isDefined`

isDefined

Class: `coder.descriptor.TypedRegion`

Namespace: `coder.descriptor`

Determine if data element is defined in generated code

Syntax

```
tf = isDefined(implementation)
```

Description

`tf = isDefined(implementation)` returns 1 (true) if the data element is defined by the generated code. Otherwise, it returns 0 (false).

The data element is defined if all members of the `TypedRegion` object exist in the generated code.

Input Arguments

implementation — Description of implementation of data

`coder.descriptor.TypedRegion` object

Description of implementation of data in the generated code, specified as a `coder.descriptor.TypedRegion` object.

Output Arguments

tf — True or false result

1 | 0

True or false result, returned as a 1 or 0 of data type `logical`.

Data Types: `logical`

Examples

Determine if Inport Block Is Defined by Generated Code

- 1 Open the model.

```
openExample('CustomCodeComments')
```
- 2 Build the model.

```
slbuild('CustomCodeComments')
```
- 3 Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments')
```
- 4 Return properties of the Inports data interface type in the generated code.

```
dataInterfaces = getDataInterfaces(codeDescObj, 'Inports')
```

- 5 Return properties of the code representation of the first Inport block.
- 6 Return whether the data element is defined by its code representation.

```
implementation = dataInterfaces(1).Implementation
```

```
isDefined(implementation)
```

```
ans =  
  logical  
  1
```

Since `isDefined` returned logical 1, all members of `implementation` exist in the generated code.

Version History

Introduced in R2025a

See Also

`coder.descriptor.TypedRegion` | `getAddress` | `getExpression`

`coder.mapping.api.CodeMapping`

Model data and function interface configuration for C code generation

Description

A code mappings object and related functions enable you to configure C code generation for data and functions of a Simulink model. For model data elements, code mappings associate data elements with configurations that consist of a storage class and storage class properties. For functions, code mappings associate entry-point functions with configurations that consist of a function customization template. Reduce the effort of preparing a model for C code generation by specifying default configurations for categories of data elements and functions across a model. Override default configurations by configuring data elements or functions individually. For smaller models, you can choose to configure each data element and function individually. For models with an attached Embedded Coder Dictionary, you can map model elements to service interfaces defined in that dictionary.

Creation

When you select a code generation app from the Apps tab in the Simulink Editor, such as the **Simulink Coder** or **Embedded Coder** app, the app creates a `coder.mapping.api.CodeMapping` object if code mappings do not already exist. The app creates code mappings based on code customization settings stored in the model active configuration set object. The configuration set object can specify memory sections for data and functions.

Create a `coder.mapping.api.CodeMapping` object programmatically by calling the function `coder.mapping.utils.create`. Create a mapping based on the active configuration set object or based on the default memory section and shared utility naming rule configurations of another configuration set object.

Object Functions

<code>addSignal</code>	Add block output signal to model code mappings
<code>coder.mapping.api.get</code>	Get code mappings for model
<code>coder.mapping.utils.create</code>	Create code mappings object for configuring data and function interface for C and C++ code generation
<code>find</code>	Get model elements for the category of model code mappings
<code>getDataDefault</code>	Get default storage class or storage class property setting for model data category
<code>getDataStore</code>	Get code and calibration configuration from code mappings for local or shared local data store
<code>getDataTransfer</code>	Get code configuration from code mappings for signal representing data transfer
<code>getDeploymentType</code>	Get deployment type of model
<code>getFunction</code>	Get code configuration from code mappings for model function
<code>getFunctionDefault</code>	Get default function customization template or memory section for model functions category

<code>getInport</code>	Get code and calibration configuration from code mappings for root-level inport
<code>getModelParameter</code>	Get code and calibration configuration from code mappings for model parameters
<code>getModelVariantControl</code>	Get code configuration from code mappings for Simulink.VariantControl object
<code>getModelVariantVariable</code>	Get code configuration from code mappings for Simulink.VariantVariable object
<code>getOutport</code>	Get code and calibration configuration from code mappings for root-level outport
<code>getSignal</code>	Get code and calibration configuration from code mappings for block output signal
<code>getState</code>	Get code and calibration configuration from code mappings for block state
<code>removeSignal</code>	Remove block output signal from model code mappings
<code>setDataDefault</code>	Set default storage class and storage class property values for model data category
<code>setDataStore</code>	Configure local or shared local data store for code and calibration file (a2l) generation
<code>setDataTransfer</code>	Set code configuration in code mappings for signal representing data transfer
<code>setDeploymentType</code>	Set deployment type of model
<code>setFunction</code>	Set coder mapping information for model function
<code>setFunctionDefault</code>	Set default function customization template and memory section for model functions category
<code>setInport</code>	Configure root-level inports for code and calibration file (a2l) generation
<code>setModelParameter</code>	Configure model parameter for code and calibration file (a2l) generation
<code>setModelVariantControl</code>	Configure Simulink.VariantControl object for code generation
<code>setOutport</code>	Configure root-level outport for code and calibration file (a2l) generation
<code>setSignal</code>	Configure block signal data for code and calibration file (a2l) generation
<code>setState</code>	Configure block states for code and calibration file (a2l) generation

Examples

Programmatically Create and Use Code Mappings of Simulink Models

Use the programmatic interface to create and use code mappings objects of Simulink models.

Load the model `NoCoderMapping`.

```
simulinkModel = "NoCoderMapping";
load_system(simulinkModel);
```

Use a `try-catch` block to determine whether a code mappings object exists for the model. Inside the `try` block, attempt to retrieve the existing object by using function `coder.mapping.api.get`. Inside the `catch` block, create a new code mappings object by using the function `coder.mapping.utils.create`. Store the code mappings object in the variable `codeMapObj`.

Add print messages to show whether the model had an existing code mappings object or not.

```
try
    codeMapObj = coder.mapping.api.get(simulinkModel);
```

```

fprintf("'" + ...
" =====\n" + ...
" The model already had code mappings.\n" + ...
" =====\n");
catch
fprintf("'" + ...
" =====\n" + ...
" The model does not have code mappings.\n" + ...
" Creating new code mappings for the model.\n" + ...
" =====\n");
codeMapObj = coder.mapping.utils.create(simulinkModel);
end

```

```

=====
The model does not have code mappings.
Creating new code mappings for the model.
=====

```

Retrieve the storage class of input port `inport_1` of the model.

```
getInport(codeMapObj,"inport_1","StorageClass")
```

```
ans =
'Auto'
```

Set the storage class of input port `inport_1` to `ExportedGlobal`.

```
setInport(codeMapObj,"inport_1",StorageClass="ExportedGlobal")
```

Use the same try-catch block to see how the function `coder.mapping.api.get` is now able to retrieve the existing code mappings of the model.

```

try
codeMapObj = coder.mapping.api.get(simulinkModel);
fprintf("'" + ...
" =====\n" + ...
" The model already had code mappings.\n" + ...
" =====\n");
catch
fprintf("'" + ...
" =====\n" + ...
" The model does not have code mappings.\n" + ...
" Creating new code mappings for the model.\n" + ...
" =====\n");
codeMapObj = coder.mapping.utils.create(simulinkModel);
end

```

```

=====
The model already had code mappings.
=====

```

Retrieve the storage class of input port `inport_1` of the model. Notice that it is the storage class that you set previously.

```
getInport(codeMapObj,"inport_1","StorageClass")
```

```
ans =
'ExportedGlobal'
```

Close the model without saving it.

```
close_system(simulinkModel,false)
```

Programmatically Get and Use Multiple Code Mapping Types with Simulink® Models

Use the programmatic interface to retrieve and use different types of code mappings objects of a Simulink model.

Load the model `CoderMapAPI`.

```
codeMapModel = "CoderMapAPI";
load_system(codeMapModel);
```

Use the function `coder.mapping.api.get` to retrieve and store the Simulink® Coder™ and the Embedded Coder® mapping objects of this model.

```
simCodeMapObj = coder.mapping.api.get(codeMapModel,"SimulinkCoderC");
eCodeMapObj = coder.mapping.api.get(codeMapModel,"EmbeddedCoderC");
```

Store all root-level input port names.

```
in_port_names = "in_port_"+(1:4) '
in_port_names = 4x1 string
    "in_port_1"
    "in_port_2"
    "in_port_3"
    "in_port_4"
```

For each of the code mapping objects, define an inline function that sets the storage class of the input ports. Use the `arrayfun` function to set the storage class of all root-level input ports of the Simulink code mappings object to `ImportedExtern` and set the storage class of the root-level input ports of the Embedded Coder mappings object to `ImportedExternPointer`.

```
configSimCoderPortStoreClassFcn = @(port_name) setInport(simCodeMapObj,port_name,StorageClass="ImportedExtern");
configECoderPortStoreClassFcn = @(port_name) setInport(eCodeMapObj,port_name,StorageClass="ImportedExternPointer");
arrayfun(configSimCoderPortStoreClassFcn,in_port_names);
arrayfun(configECoderPortStoreClassFcn,in_port_names);
```

For each of the code mapping objects, define an inline function that displays the storage class of the input ports.

```
dispSimCoderPortStoreClassFcn = @(port_name) disp(getInport(simCodeMapObj,port_name,"StorageClass"));
dispECoderPortStoreClassFcn = @(port_name) disp(getInport(eCodeMapObj,port_name,"StorageClass"));
```

Display the storage class of the root-level input ports for each of the code mappings objects.

```
arrayfun(dispSimCoderPortStoreClassFcn,in_port_names);
```

```
ImportedExtern
ImportedExtern
ImportedExtern
ImportedExtern
```

```
arrayfun(dispatchCoderPortStoreClassFcn,in_port_names);
```

```
ImportedExternPointer
ImportedExternPointer
ImportedExternPointer
ImportedExternPointer
```

Root-level input ports with `ImportedExtern` and `ImportedExternPointer` storage classes are declared in the generated private header file of the model, in separate sections. Store the header file names of the two system target files so that you can observe these sections in both of header files.

```
priv_h_simcoder_file = fullfile(codeMapModel+"_grt_rtw",codeMapModel+"_private.h")
```

```
priv_h_simcoder_file =
"CoderMapAPI_grt_rtw\CoderMapAPI_private.h"
```

```
priv_h_ecoder_file = fullfile(codeMapModel+"_ert_rtw",codeMapModel+"_private.h")
```

```
priv_h_ecoder_file =
"CoderMapAPI_ert_rtw\CoderMapAPI_private.h"
```

Set the model target file to `grt.tlc` and generate code from the model.

```
set_param(codeMapModel,SystemTargetFile="grt.tlc")
evalc("slbuild(codeMapModel)");
```

These are the declarations of the root-level input ports in the `grt.tlc` header file, `priv_h_simcoder_file`:

```
/* Imported (extern) block signals */
extern real_T in_port_1;          /* '<Root>/in_port_1' */
extern real_T in_port_2;          /* '<Root>/in_port_2' */
extern real_T in_port_3;          /* '<Root>/in_port_3' */
extern real_T in_port_4;          /* '<Root>/in_port_4' */
```

The root-level input ports are declared in the `ImportedExtern` section.

To open the header file, enter this command in the MATLAB® Command Window:

```
edit(priv_h_simcoder_file)
```

Now set the model target file to `ert.tlc` and generate code from the model.

```
set_param(codeMapModel,SystemTargetFile="ert.tlc")
evalc("slbuild(codeMapModel)");
```

These are the declarations of the root-level input ports in the `ert.tlc` header file, `priv_h_ecoder_file`:

```
/* Imported (extern) pointer block signals */
extern real_T *in_port_1;         /* '<Root>/in_port_1' */
extern real_T *in_port_2;         /* '<Root>/in_port_2' */
extern real_T *in_port_3;         /* '<Root>/in_port_3' */
extern real_T *in_port_4;         /* '<Root>/in_port_4' */
```

The root-level input ports in this header file are declared in the `ImportedExternPointer` section.

To open the header file, enter this command in the MATLAB Command Window:

```
edit(priv_h_ecoder_file)
```

Version History

Introduced in R2020b

R2023a: Automatic Code Suggestions and Completions

Starting in R2023a, tab completion is available for function names and arguments and for object properties. After you enter the first few characters of a function, input argument, or object property, press the **Tab** key to let MATLAB automatically complete the typing. MATLAB adds the remaining characters of the function, argument, or parameter. If you did not enter anything or if there are multiple options that begin with the characters you entered, MATLAB opens a list of available alternatives you can choose from. To learn more about tab completion, see [Code Suggestions and Completions](#).

See Also

[coder.mapping.api.get](#) | [coder.mapping.utils.create](#) |
[coder.mapping.api.CoderDictionary](#) | [coder.mapping.api.CodeMappingCPP](#)

Topics

[“C Data Code Interface Configuration for Model Interface Elements”](#)

[“Programmatically Configure C++ Interface”](#)

[“Service Interfaces”](#)

addSignal

Add block output signal to model code mappings

Syntax

```
addSignal(coderMapObj,portHandle)
addSignal(coderMapObj,portHandle,Name=Value)
```

Description

`addSignal(coderMapObj,portHandle)` adds signals specified by the block output port handles to the specified model code mappings.

This function does not apply to signals that originate from root-level Inport blocks.

`addSignal(coderMapObj,portHandle,Name=Value)` specifies one or more name-value arguments in addition to the input argument in the previous syntax.

Examples

Programmatically Add and Remove Signal Lines in Code Mappings of Simulink® Models

Use the programmatic interface to add and remove signals in the code mapping configuration of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure it is open with the **Signals/States** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model `CoderMapAPI`.

```
simulinkModel = "CoderMapAPI";
open_system(simulinkModel);
```

Use the function `coder.mapping.api.get` to retrieve the code mapping object of the model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Get the handles of the output ports of the four Gain blocks and store them in variables.

```
port_handle_1 = get_param(simulinkModel+"/gain_1","PortHandles").Outputport;
port_handle_2 = get_param(simulinkModel+"/gain_2","PortHandles").Outputport;
port_handle_3 = get_param(simulinkModel+"/gain_3","PortHandles").Outputport;
port_handle_4 = get_param(simulinkModel+"/gain_4","PortHandles").Outputport
```

```
port_handle_4 =
305.0021
```

Use the function `addSignal` to add the signal to the code mappings of the model. Set the storage class of `sig_1` and `sig_3` to `ImportedExtern` and the storage class of `sig_2` and `sig_4` to `ImportedExternPointer`. Update the model.

```
addSignal(codeMapObj,[port_handle_1,port_handle_3],StorageClass="ImportedExtern")
addSignal(codeMapObj,[port_handle_2,port_handle_4],StorageClass="ImportedExternPointer")
set_param(simulinkModel,"SimulationCommand","update")
```

Use the `slbuild` command to generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Signals with `ImportedExtern` and `ImportedExternPointer` storage classes are declared in separate sections of the generated private header file of the model. Store the name of the private header file in the variable `priv_h_file`.

```
priv_h_file = fullfile(simulinkModel+"_grt_rtw",simulinkModel+"_private.h")
```

```
priv_h_file =
"CoderMapAPI_grt_rtw\CoderMapAPI_private.h"
```

The declarations of the signals appear as the following code in the header file:

```
/* Imported (extern) block signals */
extern real_T sig_1;          /* '<Root>/gain_1' */
extern real_T sig_3;          /* '<Root>/gain_3' */
extern real_T in_port_1;      /* '<Root>/in_port_1' */
extern real_T in_port_2;      /* '<Root>/in_port_2' */

/* Imported (extern) pointer block signals */
extern real_T *sig_2;         /* '<Root>/gain_2' */
extern real_T *sig_4;         /* '<Root>/gain_4' */
extern real_T *in_port_3;     /* '<Root>/in_port_3' */
extern real_T *in_port_4;     /* '<Root>/in_port_4' */
```

The signals are defined according to the specified storage classes.

To open the header file, enter this command in the MATLAB® Command Window.

```
edit(priv_h_file)
```

Use the function `removeSignal` to remove the first two signal lines from the code mappings of the model. Update the model.

```
removeSignal(codeMapObj,[port_handle_1,port_handle_2])
set_param(simulinkModel,"SimulationCommand","update")
```

Rebuild the model with the `slbuild` command.

```
evalc("slbuild(simulinkModel)");
```

The updated sections appear as the following code in the header file.

```
/* Imported (extern) block signals */
extern real_T sig_3;          /* '<Root>/gain_3' */
extern real_T in_port_1;      /* '<Root>/in_port_1' */
extern real_T in_port_2;      /* '<Root>/in_port_2' */

/* Imported (extern) pointer block signals */
extern real_T *sig_4;         /* '<Root>/gain_4' */
extern real_T *in_port_3;     /* '<Root>/in_port_3' */
extern real_T *in_port_4;     /* '<Root>/in_port_4' */
```

The removed signals no longer appear in the header file.

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

portHandle — Signal to add to code mappings

block output port handle | port handle array

Signal to add to the code mappings, specified as the output port handle of the source block of the signal. To add multiple signals simultaneously, specify an output port handle array.

Example: `addSignal(coderMapObj,[outport_1_Handle,outport_2_Handle])`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

"Auto" | "ExportedGlobal" | ...

The name of the storage class to set for the specified signals. The storage class is either defined in the internal Embedded Coder Dictionary of the model, or in a shared Embedded Coder Dictionary that is attached to the model.

To learn more about signal data configuration for code generation, see “Configure Signal Data for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the signal data in the generated code.

Data Types: `char` | `string`

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

GetFunction — Name of get function

character vector | string scalar

Name of a **get** function that a data element calls in the generated code. Applies to storage class **GetSet**.

Data Types: `char` | `string`

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes **ExportToFile**, **GetSet**, **ImportFromFile**, and **Volatile**.

Data Types: `char` | `string`

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data that is used by other models in the same model hierarchy. The code generated for the owner model includes the global data definition. Applies to storage classes **ExportToFile** and **Volatile**.

Data Types: `char` | `string`

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

`True` | `False`

When model configuration parameter **Array layout** is set to **Row-major**, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes **ExportToFile**, **GetSet**, **ImportFromFile**, **Localizable**, and **Volatile**.

Data Types: `logical`

SetFunction — Name of set function

character string | string scalar

Name of a **set** function that a data element calls in the generated code. Applies to storage class **GetSet**.

Data Types: `char` | `string`

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes **Bitfield** and **Struct**.

Data Types: `char` | `string`

storageClassPropertyName — Value of storage class property

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

MeasurementService — Name of measurement service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a measurement service interface defined in the Embedded Coder Dictionary. By configuring the measurement service interface for signals, states, and data stores, you can preserve the data in the generated code for measurement. To use the dictionary default, specify "Dictionary default". If data from the state does not need to be preserved in the code, specify "Not measured".

To configure the measurement service interface, the Embedded Coder Dictionary that defines a service interface configuration must be attached to the model. For more information, see "Configure Measurement Service Interfaces for Signals, States, and Data Stores".

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getSignal` | `removeSignal` | `setDataDefault` | `setSignal`

Topics

"Configure Signal Data for C Code Generation"

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Measurement Service Interfaces for Signals, States, and Data Stores"

find

Namespace: coder.mapping.api

Get model elements for the category of model code mappings

Syntax

```
modelElementsFound = find(coderMapObj,elementCategory)
modelElementsFound= find(coderMapObj,elementCategory,Name=Value)
```

Description

`modelElementsFound = find(coderMapObj,elementCategory)` returns the elements in the model code mappings of the specified category as an array of objects.

`modelElementsFound= find(coderMapObj,elementCategory,Name=Value)` returns the elements in the model code mappings of the specified category that match specified property and value criteria.

Examples

Programmatically Find Code Mapping Elements of Simulink® Models

Use the programmatic interface to find model elements that are related to the code mappings.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure the **Code Mappings** editor is open with the **Inports** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model `CoderMapAPI`.

```
simulinkModel = "CoderMapAPI";
open_system(simulinkModel);
```

Retrieve the code mappings object of this model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Find input port handles of ports with `ImportedExtern` storage class and input ports with `ImportedExternPointer` storage class.

```
importedExternInportHandles = find(codeMapObj,"Inports",StorageClass="ImportedExtern")'
```

```
importedExternInportHandles = 2x1
```

```
    151.0038
    153.0037
```

```
importedExternPointerInportHandles = find(codeMapObj,"Inports",StorageClass="ImportedExternPointe
```

```
importedExternPointerInportHandles = 2x1
```

```
155.0032  
157.0024
```

Get the names of the input ports.

```
importedExterInportNames = string(get_param(importedExternInportHandles, "Name"))
```

```
importedExterInportNames = 2x1 string  
"in_port_1"  
"in_port_2"
```

```
importedExternPointerInportNames = string(get_param(importedExternPointerInportHandles, "Name"))
```

```
importedExternPointerInportNames = 2x1 string  
"in_port_3"  
"in_port_4"
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Root-level ports with `ImportedExtern` and `ImportedExternPointer` storage classes are declared in the generated private header file of the model in separate sections.

Store the name of the private header file.

```
priv_h_file = fullfile(simulinkModel+"_grt_rtw",simulinkModel+"_private.h")
```

```
priv_h_file =  
"CoderMapAPI_grt_rtw\CoderMapAPI_private.h"
```

These are the declarations of the root-level ports in the header file:

```
/* Imported (extern) block signals */  
extern real_T in_port_1;          /* '<Root>/in_port_1' */  
extern real_T in_port_2;          /* '<Root>/in_port_2' */  
  
/* Imported (extern) pointer block signals */  
extern real_T *in_port_3;         /* '<Root>/in_port_3' */  
extern real_T *in_port_4;         /* '<Root>/in_port_4' */
```

The storage classes of the ports match the storage classes passed to the `find` function.

To open the header file, enter this command in the MATLAB® Command Window.

```
edit(priv_h_file)
```

To see the `ImportedExtern` inports highlighted in the model canvas and in the **Code Mappings** editor, enter this command:

```
arrayfun(@(pHandle) set_param(pHandle,Selected="on"),importedExternInportHandles);
```

Programmatically Find Entry-Point Functions with Function Customization Template Mapped to Model Default

Use the programmatic interface to find entry-point functions with function customization templates mapped to the model default, and remap templates to the default customization template.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure the **Code Mappings** editor is open with the **Functions** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model ECoderMapAPI.

```
simulinkModel = "ECoderMapAPI";
open_system(simulinkModel);
```

Retrieve the code mappings object of the model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Use the code mappings object to find all entry-point functions with function customization template mapped to the model default. Then display the names of the functions you found.

```
funcObjs = find(codeMapObj,"Functions",FunctionCustomizationTemplate="Model default");
disp(funcObjs')

"Initialize"
"Terminate"
"Periodic:D1"
```

Map the function customization template of the functions you found to Default.

```
setFunction(codeMapObj,funcObjs,FunctionCustomizationTemplate="Default");
```

Use the find function to test if there are any entry-point functions with customization template mapped to the model default. You should get an empty list.

```
find(codeMapObj,"Functions",FunctionCustomizationTemplate="Model default")

ans =

0x0 empty string array
```

Map the function customization template of the functions you found back to model default.

```
setFunction(codeMapObj,funcObjs,FunctionCustomizationTemplate="Model Default");
```

Input Arguments

coderMapObj — Code mapping object
CodeMapping object

Code mapping object returned by a call to function `coder.mapping.api.get`

Example: `coderMapObj = coder.mapping.api.get(bdroot)`

elementCategory — Model element category
"DataStores" | "DataTransfers" | ...

Model element category that you search for in the model code mappings, specified as one of these values:

- "DataStores"
- "DataTransfers"
- "ExportedFunctions"
- "Functions"
- "FunctionCallers"
- "Inports"
- "ModelParameters"
- "ModelParameterArguments"
- "ModelVariantControls"
- "ModelVariantControlArguments"
- "ModelVariantVariables"
- "Outports"
- "PartitionFunctions"
- "PartitionUpdateFunctions"
- "PeriodicFunctions"
- "PeriodicUpdateFunctions"
- "ResetFunctions"
- "Signals"
- "SimulinkFunctions"
- "States"

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `find(coderMapObj,StorageClass="Bitfield")`

Data Interfaces

StorageClass — Name of storage class

"Auto" | "ExportedGlobal" | ...

The name of data element storage class to include in the search criteria. The storage class is either defined in the internal Embedded Coder Dictionary of the model, or in a shared Embedded Coder Dictionary that is attached to the model.

The list of available storage classes vary depending on the specified category.

Identifier — Code identifier

character vector | string scalar

Name that the code generator uses to identify a data element in generated code. Applies to storage classes other than "Auto".

Data Types: char | string

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Data Types: char | string

FunctionCustomizationTemplate — Name of function customization template

character vector | string scalar

Name of a function customization template for a model that is defined in the Embedded Coder Dictionary.

Data Types: char | string

FunctionName — Name of entry-point function

character vector | string scalar

Name of an entry-point function generated for a model.

Data Types: char | string

GetFunction — Name of get function

character vector | string scalar

Name of a get function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, Define, ExportToFile, GetSet, ImportedDefine, ImportFromFile, and Volatile.

Data Types: char | string

MemorySection — Name of memory section

character vector | string scalar

Name of a memory section for a model that is defined in the Embedded Coder Dictionary.

Data Types: char | string

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the owner model includes the global data definition. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes Const, ConstVolatile, ExportToFile, FileScope, GetSet, ImportFromFile, Localizable, and Volatile.

Data Types: logical

SetFunction — Name of set function

character string | string scalar

Name of a set function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Data Types: char | string

storageClassPropertyName — Value of storage class property

depends on property definition

Storage class property defined in the model Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Service Interfaces**ReceiverService — Name of receiver service interface**

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a receiver service interface defined in the Embedded Coder Dictionary. Within a target environment, a component receives data from other components by calling the target platform receiver service. To use the dictionary default, specify "Dictionary default".

To configure the receiver service interface, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model. For more information, see "Configure Sender and Receiver Service Interfaces for Model Inports and Outports".

Data Types: char | string

SenderService — Name of sender service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a sender service interface defined in the Embedded Coder Dictionary. Within a target environment, a component sends data to other components by calling the target platform sender service. To use the dictionary default, specify "Dictionary default".

To configure the sender service interface, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model. For more information, see “Configure Sender and Receiver Service Interfaces for Model Inports and Outports”.

Data Types: `char` | `string`

MeasurementService — Name of measurement service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a measurement service interface defined in the Embedded Coder Dictionary. By configuring the measurement service interface for signals, states, and data stores, you can preserve the data in the generated code for measurement. To use the dictionary default, specify "Dictionary default". If data from the state does not need to be preserved in the code, specify "Not measured".

To configure the measurement service interface, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model. For more information, see “Configure Measurement Service Interfaces for Signals, States, and Data Stores”.

Data Types: `char` | `string`

ParameterTuningService — Name of parameter tuning service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a parameter tuning service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify "Dictionary default".

To configure the parameter tuning service interface, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model. For more information, see “Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments”.

Data Types: `char` | `string`

ParameterArgumentTuningService — Name of parameter argument tuning service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a parameter argument tuning service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify "Dictionary default".

To configure the parameter argument tuning service interface, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model. For more information, see “Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments”.

Data Types: `char` | `string`

TimerService — Name of timer service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a timer service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify "Dictionary default".

This property is only applicable for exported functions.

To configure the timer service interface, an Embedded Coder Dictionary that defines a service interface configuration must be attached to the model. For more information, see “Configure Timer Service Interfaces”.

Data Types: `char` | `string`

Output Arguments

modelElementsFound — Model elements found

`handle array` | `string array`

Model elements found, returned as a string array or a handle array, depending on the specified element category.

Element Category	Type of Object Returned
"Inports", "Outports", and "States"	Block handle
"Signals"	Port handle
"DataStores"	Block handle
"ModelParameters"	Model parameter name
"ModelParameterArguments"	Model parameter argument name
"ModelVariantControls"	<code>Simulink.VariantControl</code> object name
"ModelVariantControlArguments"	<code>Simulink.VariantControl</code> model argument name
"ModelVariantVariables"	<code>Simulink.VariantVariable</code> object name
"ExportedFunctions", "Functions", "PartitionFunctions", "PartitionUpdateFunctions", "PeriodicFunctions", "PeriodicUpdateFunctions", "ResetFunctions", and "SimulinkFunctions"	Function name

Version History

Introduced in R2020b

R2025a: Support for `Simulink.VariantControl` and `Simulink.VariantVariable` objects

The `find` function now supports these values for the `elementCategory` argument to find `Simulink.VariantControl` and `Simulink.VariantVariable` objects in the code mapping categories.

- `ModelVariantControls` — Find `Simulink.VariantControl` objects present in the model code mappings
- `ModelVariantControlArguments` — Find `Simulink.VariantControl` objects set as model arguments and present in the model code mappings
- `ModelVariantVariables` — Find `Simulink.VariantVariable` objects present in the model code mappings

R2022b: Model parameters and parameter arguments returned separately by find function
Behavior changed in R2022b

The `find` function now returns model parameter arguments separately from model parameters.

Starting in R2022b, to return all elements in the model code mappings that are model parameter arguments, enter the following.

```
cm = coder.mapping.api.get("myConfigModel");  
modelParamArgs = find(cm,"ModelParameterArguments");
```

To return all elements in the model code mappings that are model parameters, enter the following.

```
cm = coder.mapping.api.get("myConfigModel");  
modelParams = find(cm,"ModelParameters");
```

In previous releases, specifying `ModelParameters` as the `elementCategory` argument returned both model parameters and model parameter arguments.

See Also

`coder.mapping.api.get`

Topics

"C Data Code Interface Configuration for Model Interface Elements"

coder.mapping.api.get

Get code mappings for model

Syntax

```
coderMapObj = coder.mapping.api.get(simulinkModel)
coderMapObj = coder.mapping.api.get(simulinkModel,codeMappingType)
coderDataDictObj = coder.mapping.api.get(coderDictionary)
```

Description

`coderMapObj = coder.mapping.api.get(simulinkModel)` returns the active code mappings for the specified model as object `coderMapObj`. Code mappings associate model data elements and functions with configurations for C or C++ code generation. If a model has multiple code mappings, the active code mappings are the mappings associated with the active system target file.

If code mappings do not exist, an error is generated. Simulink creates a code mappings object when you open a coder app for the model. If you have not opened a coder app for the model, you can invoke the `coder.mapping.utils.create` function to create a code mappings object.

`coderMapObj = coder.mapping.api.get(simulinkModel,codeMappingType)` returns the code mappings for your model that correspond to the specified code mapping type as object `coderMapObj`.

`coderDataDictObj = coder.mapping.api.get(coderDictionary)` returns the object `coderDataDictObj` that you can use to get and set data and function defaults in the shared dictionary specified by `coderDictionary`.

Note

- To save changes you make in the dictionary with the object `coderMapObj`, create and use a general dictionary object `Simulink.data.Dictionary` for the same dictionary.
 - This syntax of the function is only supported for data interface dictionaries.
-

Examples

Programmatically Create and Use Code Mappings of Simulink Models

Use the programmatic interface to create and use code mappings objects of Simulink models.

Load the model `NoCoderMapping`.

```
simulinkModel = "NoCoderMapping";
load_system(simulinkModel);
```

Use a try-catch block to determine whether a code mappings object exists for the model. Inside the try block, attempt to retrieve the existing object by using function `coder.mapping.api.get`.

Inside the catch block, create a new code mappings object by using the function `coder.mapping.utils.create`. Store the code mappings object in the variable `codeMapObj`.

Add print messages to show whether the model had an existing code mappings object or not.

```
try
    codeMapObj = coder.mapping.api.get(simulinkModel);
    fprintf("'" + ...
        " =====\n" + ...
        " The model already had code mappings.\n" + ...
        " =====\n");
catch
    fprintf("'" + ...
        " =====\n" + ...
        " The model does not have code mappings.\n" + ...
        " Creating new code mappings for the model.\n" + ...
        " =====\n");
    codeMapObj = coder.mapping.utils.create(simulinkModel);
end

=====
The model does not have code mappings.
Creating new code mappings for the model.
=====
```

Retrieve the storage class of input port `inport_1` of the model.

```
getInport(codeMapObj,"inport_1","StorageClass")
```

```
ans =
'Auto'
```

Set the storage class of input port `inport_1` to `ExportedGlobal`.

```
setInport(codeMapObj,"inport_1",StorageClass="ExportedGlobal")
```

Use the same try-catch block to see how the function `coder.mapping.api.get` is now able to retrieve the existing code mappings of the model.

```
try
    codeMapObj = coder.mapping.api.get(simulinkModel);
    fprintf("'" + ...
        " =====\n" + ...
        " The model already had code mappings.\n" + ...
        " =====\n");
catch
    fprintf("'" + ...
        " =====\n" + ...
        " The model does not have code mappings.\n" + ...
        " Creating new code mappings for the model.\n" + ...
        " =====\n");
    codeMapObj = coder.mapping.utils.create(simulinkModel);
end

=====
The model already had code mappings.
=====
```

Retrieve the storage class of input port `inport_1` of the model. Notice that it is the storage class that you set previously.

```
getInport(codeMapObj, "inport_1", "StorageClass")
```

```
ans =  
'ExportedGlobal'
```

Close the model without saving it.

```
close_system(simulinkModel, false)
```

Programmatically Get and Use Code Mappings Objects with Coder Dictionaries

Programmatically get and use a code mappings object with an **Embedded Coder® Dictionary**.

To interactively observe how your commands are reflected in the **Code Mappings** editor and the model dictionary, make sure the **Code Mappings** editor is open with the **Data Defaults** tab selected, and the model dictionary is open with the **Data Defaults** section selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 . To learn how to open the model dictionary, see “Open the Embedded Coder Dictionary” on page 23-0 .

Store the model and dictionary filenames, and then open the model `ECoderMapAPI`.

```
simulinkModel = "ECoderMapAPI";  
dict_name = "ECoderDataDict.sldd";  
load_system(simulinkModel);
```

To use code mappings in a shared dictionary, configure the model to use the shared dictionary.

```
set_param(simulinkModel, EmbeddedCoderDictionary="ECoderDataDict.sldd");
```

Get to get the code mappings object of the model and use it to:

- Specify `Model default` as the storage class of all root-level ports of the model.
- Specify `Dictionary default` as the default storage class of root-level ports.

```
modelCodeMapObj = coder.mapping.api.get(simulinkModel);  
setInport(modelCodeMapObj, find(modelCodeMapObj, "Inports"), StorageClass="Model default")  
setOutport(modelCodeMapObj, find(modelCodeMapObj, "Outports"), StorageClass="Model default")  
setDataDefault(modelCodeMapObj, "Inports", StorageClass="Dictionary default")  
setDataDefault(modelCodeMapObj, "Outports", StorageClass="Dictionary default")
```

A coder dictionary contains, among other things, definitions that can be used by the code mappings of the model. You can use the code mapping API to retrieve and control the data and function defaults in shared coder dictionaries.

Get the code mappings object of the dictionary, `ECoderDataDict.sldd`. You use this object to view and control the data and function defaults of the dictionary.

```
coderMapDictObj = coder.mapping.api.get(dict_name);
```

Get a general dictionary object for the same shared dictionary. You use this object to control general elements of the dictionary, that are not related to code mappings. In this example, you need this

general dictionary object to save the dictionary after you make changes to it with the code mapping object of the dictionary.

```
dictObj = Simulink.data.dictionary.open(dict_name);
```

Use the code mappings object of the dictionary to set the default storage class of input ports to `Localizable` and the default storage class of output ports to `Volatile`.

```
setDataDefault(coderMapDictObj,"Inports",StorageClass="Localizable");
setDataDefault(coderMapDictObj,"Outports",StorageClass="Volatile");
```

Verify that the default storage classes are set as you intended.

```
getDataDefault(coderMapDictObj,"Inports","StorageClass")
```

```
ans =
'Localizable'
```

```
getDataDefault(coderMapDictObj,"Outports","StorageClass")
```

```
ans =
'Volatile'
```

`Localizable` and `Volatile` variables are declared in separate sections of the generated model header file. Store the header file name in the variable `model_h_file`.

```
model_h_file = fullfile(simulinkModel+"_ert_rtw",simulinkModel+".h")
```

```
model_h_file =
"ECoderMapAPI_ert_rtw\ECoderMapAPI.h"
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

These are the declarations of the root-level input and output ports in the header file:

```
/* Declaration for custom storage class: Localizable */
extern real_T in_port_1;          /* '<Root>/in_port_1' */
extern real_T in_port_2;          /* '<Root>/in_port_2' */
extern real_T in_port_3;          /* '<Root>/in_port_3' */
extern real_T in_port_4;          /* '<Root>/in_port_4' */

/* Declaration for custom storage class: Volatile */
extern volatile real_T out_port_1; /* '<Root>/out_port_1' */
extern volatile real_T out_port_2; /* '<Root>/out_port_2' */
extern volatile real_T out_port_3; /* '<Root>/out_port_3' */
extern volatile real_T out_port_4; /* '<Root>/out_port_4' */
```

Ports are of the storage classes set in the data defaults of the dictionary.

To open the header file enter this command in the MATLAB® Command Window:

```
edit(model_h_file)
```

Use the `setDataDefault` function with the code mapping object of the dictionary to switch between the default storage classes of the input and output ports in the dictionary.

```
setDataDefault(coderMapDictObj,"Inports",StorageClass="Volatile");
setDataDefault(coderMapDictObj,"Outports",StorageClass="Localizable");
```

Verify that the default storage classes are set as you intended.

```
getDataDefault(coderMapDictObj, "Inports", "StorageClass")
```

```
ans =  
'Volatile'
```

```
getDataDefault(coderMapDictObj, "Outports", "StorageClass")
```

```
ans =  
'Localizable'
```

Generate code from the model again.

```
evalc("slbuild(simulinkModel)");
```

The declarations of the input and output ports have been updated in the header file with the storage classes you configured them to be.

```
/* Declaration for custom storage class: Localizable */  
extern real_T out_port_1;          /* '<Root>/out_port_1' */  
extern real_T out_port_2;          /* '<Root>/out_port_2' */  
extern real_T out_port_3;          /* '<Root>/out_port_3' */  
extern real_T out_port_4;          /* '<Root>/out_port_4' */
```

```
/* Declaration for custom storage class: Volatile */  
extern volatile real_T in_port_1;   /* '<Root>/in_port_1' */  
extern volatile real_T in_port_2;   /* '<Root>/in_port_2' */  
extern volatile real_T in_port_3;   /* '<Root>/in_port_3' */  
extern volatile real_T in_port_4;   /* '<Root>/in_port_4' */
```

Use the general dictionary object, `dictObj`, to save the changes you made to the dictionary and to close it.

```
saveChanges(dictObj);  
close(dictObj);
```

Programmatically Get and Use Multiple Code Mapping Types with Simulink® Models

Use the programmatic interface to retrieve and use different types of code mappings objects of a Simulink model.

Load the model `CoderMapAPI`.

```
codeMapModel = "CoderMapAPI";  
load_system(codeMapModel);
```

Use the function `coder.mapping.api.get` to retrieve and store the Simulink® Coder™ and the Embedded Coder® mapping objects of this model.

```
simCodeMapObj = coder.mapping.api.get(codeMapModel, "SimulinkCoderC");  
eCodeMapObj = coder.mapping.api.get(codeMapModel, "EmbeddedCoderC");
```

Store all root-level input port names.

```
in_port_names = "in_port_" + (1:4)'
```



```
in_port_names = 4x1 string
    "in_port_1"
    "in_port_2"
    "in_port_3"
    "in_port_4"
```

For each of the code mapping objects, define an inline function that sets the storage class of the input ports. Use the `arrayfun` function to set the storage class of all root-level input ports of the Simulink code mappings object to `ImportedExtern` and set the storage class of the root-level input ports of the Embedded Coder mappings object to `ImportedExternPointer`.

```
configSimCoderPortStoreClassFcn = @(port_name) setInport(simCodeMapObj,port_name,StorageClass="ImportedExtern");
configECoderPortStoreClassFcn = @(port_name) setInport(eCodeMapObj,port_name,StorageClass="ImportedExternPointer");
arrayfun(configSimCoderPortStoreClassFcn,in_port_names);
arrayfun(configECoderPortStoreClassFcn,in_port_names);
```

For each of the code mapping objects, define an inline function that displays the storage class of the input ports.

```
dispSimCoderPortStoreClassFcn = @(port_name) disp(getInport(simCodeMapObj,port_name,"StorageClass"));
dispECoderPortStoreClassFcn = @(port_name) disp(getInport(eCodeMapObj,port_name,"StorageClass"));
```

Display the storage class of the root-level input ports for each of the code mappings objects.

```
arrayfun(dispSimCoderPortStoreClassFcn,in_port_names);
```

```
ImportedExtern
ImportedExtern
ImportedExtern
ImportedExtern
```

```
arrayfun(dispECoderPortStoreClassFcn,in_port_names);
```

```
ImportedExternPointer
ImportedExternPointer
ImportedExternPointer
ImportedExternPointer
```

Root-level input ports with `ImportedExtern` and `ImportedExternPointer` storage classes are declared in the generated private header file of the model, in separate sections. Store the header file names of the two system target files so that you can observe these sections in both of header files.

```
priv_h_simcoder_file = fullfile(codeMapModel+"_grt_rtw",codeMapModel+"_private.h")
```

```
priv_h_simcoder_file =
"CoderMapAPI_grt_rtw\CoderMapAPI_private.h"
```

```
priv_h_ecoder_file = fullfile(codeMapModel+"_ert_rtw",codeMapModel+"_private.h")
```

```
priv_h_ecoder_file =
"CoderMapAPI_ert_rtw\CoderMapAPI_private.h"
```

Set the model target file to `grt.tlc` and generate code from the model.

```
set_param(codeMapModel,SystemTargetFile="grt.tlc")
evalc("slbuild(codeMapModel)");
```

These are the declarations of the root-level input ports in the `grt.tlc` header file, `priv_h_simcoder_file`:

```
/* Imported (extern) block signals */
extern real_T in_port_1;          /* '<Root>/in_port_1' */
extern real_T in_port_2;          /* '<Root>/in_port_2' */
extern real_T in_port_3;          /* '<Root>/in_port_3' */
extern real_T in_port_4;          /* '<Root>/in_port_4' */
```

The root-level input ports are declared in the `ImportedExtern` section.

To open the header file, enter this command in the MATLAB® Command Window:

```
edit(priv_h_simcoder_file)
```

Now set the model target file to `ert.tlc` and generate code from the model.

```
set_param(codeMapModel, SystemTargetFile, "ert.tlc")
evalc("slbuild(codeMapModel)");
```

These are the declarations of the root-level input ports in the `ert.tlc` header file, `priv_h_ecoder_file`:

```
/* Imported (extern) pointer block signals */
extern real_T *in_port_1;         /* '<Root>/in_port_1' */
extern real_T *in_port_2;         /* '<Root>/in_port_2' */
extern real_T *in_port_3;         /* '<Root>/in_port_3' */
extern real_T *in_port_4;         /* '<Root>/in_port_4' */
```

The root-level input ports in this header file are declared in the `ImportedExternPointer` section.

To open the header file, enter this command in the MATLAB Command Window:

```
edit(priv_h_ecoder_file)
```

Input Arguments

simulinkModel — Simulink model

handle | character vector | string scalar

Simulink model for which to return the code mapping object, specified as a handle, or a character vector or string scalar with the model name (without the `.slx` file extension), or the relative or absolute path of the model file (including the `.slx` file extension).

Note The model must be loaded (for example, by using `load_system`).

Example: "configModel"

Data Types: char | string | handle

coderDictionary — Coder data interface dictionary

character vector | string scalar | Simulink.data.Dictionary

Data interface dictionary for which to return the code mappings object, specified as a `Simulink.data.Dictionary` object, or a character vector or string scalar with the relative or full path of the dictionary file (including the `.sldd` file extension).

Example: `"exCodeDefs.sldd"`

Data Types: `char` | `string` | `Simulink.data.Dictionary`

codeMappingType — Type of code mapping

`"SimulinkCoderC"` | `"EmbeddedCoderC"` | `"EmbeddedCoderCPP"`

The type of code mappings to return for the specified model or dictionary, specified as one of the values in the table. Code mappings enable you to associate the model with code generation configuration suitable for your platform. If the specified mappings type does not exist, an error is generated.

Code Mappings Type	Code Generation Configuration	Details
<code>"SimulinkCoderC"</code>	C language rapid prototyping platform	Simulink Coder and C language
<code>"EmbeddedCoderC"</code>	C language production platform	Embedded Coder and C language
<code>"EmbeddedCoderCPP"</code>	C ++ language production platform	Embedded Coder and C++ language

Output Arguments

coderMapObj — Code mappings

`CodeMapping` object | `CodeMappingCPP` object

The code mappings object for the model, returned as a `CodeMapping` object or a `CodeMappingCPP`.

Output	Input Object	Code Mapping Type
<code>coder.mapping.api.CodeMapping</code>	Simulink model	<code>"SimulinkCoderC"</code> or <code>"EmbeddedCoderC"</code>
<code>coder.mapping.api.CodeMappingCPP</code>	Simulink model	<code>"EmbeddedCoderCPP"</code>

coderDataDictObj — Embedded Coder data dictionary

`CoderDictionary` object

The Embedded Coder data dictionary, returned as a `coder.mapping.api.CoderDictionary` object.

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.utils.create` | `coder.mapping.api.CoderDictionary` | `coder.mapping.api.CodeMappingCPP`

Topics

"C Data Code Interface Configuration for Model Interface Elements"

"Programmatically Configure C++ Interface"

getDataDefault

Get default storage class or storage class property setting for model data category

Syntax

```
propertyValue = getDataDefault(coderMapObj,elementCategory,mapProperty)
```

Description

`propertyValue = getDataDefault(coderMapObj,elementCategory,mapProperty)` returns the value from the code mappings of the specified property for the specified data category.

You cannot specify default data interfaces for models with an attached Embedded Coder Dictionary that defines a service interface configuration.

Examples

Programmatically Get and Set Data Defaults in Code Mappings of Simulink® Models

Use the programmatic interface to get and set the data defaults in the code mappings configuration of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure the **Code Mappings** editor is open with the **Data Defaults** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model `CoderMapAPI`.

```
simulinkModel = "CoderMapAPI";
open_system(simulinkModel);
```

Retrieve the code mappings object of the model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Set the storage class of all root-level ports to `Model default` so that their generated code is determined by the value you set for `Data Defaults`.

```
setInport(codeMapObj,find(codeMapObj,"Inport"),StorageClass="Model default")
setOutport(codeMapObj,find(codeMapObj,"Outport"),StorageClass="Model default")
```

Determine the default storage classes for Input and output ports.

```
inPortDefaultStorageClass = getDataDefault(codeMapObj,"Inports","StorageClass")
```

```
inPortDefaultStorageClass =
'ImportedExtern'
```

```
outPortDefaultStorageClass = getDataDefault(codeMapObj,"Outports","StorageClass")
```

```
outPortDefaultStorageClass =
'ImportedExternPointer'
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Root-level ports with `ImportedExtern` and `ImportedExternPointer` storage classes are declared in the generated private header file of the model in separate sections.

Store the name of the header file.

```
priv_h_file = fullfile(simulinkModel+"_grt_rtw",simulinkModel+"_private.h")
```

```
priv_h_file =  
"CoderMapAPI_grt_rtw\CoderMapAPI_private.h"
```

These are the declarations of the root-level ports in the header file:

```
/* Data with Imported storage */  
extern real_T in_port_1;          /* '<Root>/in_port_1' */  
extern real_T in_port_2;          /* '<Root>/in_port_2' */  
extern real_T in_port_3;          /* '<Root>/in_port_3' */  
extern real_T in_port_4;          /* '<Root>/in_port_4' */  
  
/* Data with Imported storage (pointer) */  
extern real_T *out_port_1;        /* '<Root>/out_port_1' */  
extern real_T *out_port_2;        /* '<Root>/out_port_2' */  
extern real_T *out_port_3;        /* '<Root>/out_port_3' */  
extern real_T *out_port_4;        /* '<Root>/out_port_4' */
```

To open the header file, enter this command in the MATLAB® Command Window.

```
edit(priv_h_file)
```

Switch between the default storage class of input and output ports.

```
setDataDefault(codeMapObj,"Inports",StorageClass=outPortDefaultStorageClass);  
setDataDefault(codeMapObj,"Outports",StorageClass=inPortDefaultStorageClass);
```

Generate the code from the revised model.

```
evalc("slbuild(simulinkModel)");
```

The declarations of the root-level ports are updated in the header file according to the specified storage classes.

```
/* Data with Imported storage */  
extern real_T out_port_1;          /* '<Root>/out_port_1' */  
extern real_T out_port_2;          /* '<Root>/out_port_2' */  
extern real_T out_port_3;          /* '<Root>/out_port_3' */  
extern real_T out_port_4;          /* '<Root>/out_port_4' */  
  
/* Data with Imported storage (pointer) */  
extern real_T *in_port_1;          /* '<Root>/in_port_1' */  
extern real_T *in_port_2;          /* '<Root>/in_port_2' */  
extern real_T *in_port_3;          /* '<Root>/in_port_3' */  
extern real_T *in_port_4;          /* '<Root>/in_port_4' */
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

elementCategory — Model data element category

Constants | ExternalParameterObjects | GlobalDataStores | Inports | InternalData | ModelParameters | ModelParameterArguments | Outports | SharedLocalDataStores

Category of model data elements that you return a property value for.

Example: `"Inports"`

mapProperty — Code mapping property value to return

StorageClass | Identifier | DefinitionFile | GetFunction | HeaderFile | MemorySection | Owner | PreserveDimensions | SetFunction | StructName | storage class property name

Code mapping property that you return a value for. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary associated with the model.

Note You cannot directly retrieve or configure the default `MemorySection` for the `ModelParameterArguments` category using the code mapping object. To configure the default `MemorySection` for `ModelParameterArguments`, define a custom `StorageClass` with the `MemorySection` in a coder dictionary, and set this storage class as the default for `ModelParameterArguments`.

Information to Return	Property Name
Name of storage class	<code>StorageClass</code>
Name of variable for data element in the generated code	<code>Identifier</code>
Name of source definition file that contains definitions for global data that is read by the data element and external code	<code>DefinitionFile</code>
Name of <code>get</code> function called by code generated for the data element	<code>GetFunction</code>
Name of source header file that contains declarations for global data that is read by the model data element and external code	<code>HeaderFile</code>
Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model	<code>MemorySection</code>
Name of model for which the code generator places the definition for data element shared by multiple models in a model hierarchy	<code>Owner</code>

Information to Return	Property Name
Boolean value indicating whether code generator preserves dimensions of data that is represented as a multidimensional array	PerserveDimensions
Name of set function called by code generated for data element	SetFunction
Name of structure in generated code for data element	StructName

Example: "Identifier"

Output Arguments

propertyValue — Name of storage class or value of storage class property

character vector | string scalar | Auto | Bitfield | CompileFlag | Const | ConstVolatile | Define | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedDefine | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile

The property value is one of these values depending on the category and property that you specify.

Property	Value Returned
DefinitionFile	Character vector or string scalar that names a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.
GetFunction	Character vector or string scalar that names a get function that a data element calls in the generated code. Applies to storage class GetSet.
HeaderFile	Character vector or string scalar that names a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, Define, ExportToFile, GetSet, ImportedDefine, ImportFromFile, and Volatile.
MemorySection	Character vector or string scalar that names a memory section for a model defined in the Embedded Coder Dictionary.
Owner	Character vector or string scalar that names the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the owner model includes the global data definition. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.
PerserveDimensions	Boolean flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes Const, ConstVolatile, ExportToFile, FileScope, ImportFromFile, Localizable, and Volatile.

Property	Value Returned
SetFunction	Character vector or string scalar that names a set function, which a data element calls in the generated code. Applies to storage class GetSet.
StorageClass	One of these values: Auto, Bitfield, CompileFlag, Const, ConstVolatile, Define, Dictionary default, ExportedGlobal, ExportToFile, FileScope, GetSet, ImportedDefine, ImportedExtern, ImportedExternPointer, ImportFromFile, Localizable, Model default, Struct, Volatile
StructName	Character vector or string scalar that names that names a structure for a data element in the generated code. Applies to storage classes Bitfield and Struct.

Data Types: char | string

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `setDataDefault`

Topics

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Root-Level Inport Blocks for C Code Generation"

"Configure Root-Level Outport Blocks for C Code Generation"

"Configure Signal Data for C Code Generation"

"Configure Parameters for C Code Generation"

"Configure Block States for C Code Generation"

"Configure Data Stores for C Code Generation"

"Configure Default C Code Generation for Categories of Data Elements and Functions"

getDataStore

Get code and calibration configuration from code mappings for local or shared local data store

Syntax

```
propertyValue = getDataStore(myCodeMappingObj,dataStore,property)
```

Description

`propertyValue = getDataStore(myCodeMappingObj,dataStore,property)` returns the value of a code mapping property or calibration property for the specified local or shared local data store. For example, use this function to return the storage class or the calibration access of a measurement property configured for a local or shared local data store in a model.

Examples

Get Storage Class Configured for Local Data Store

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the name of the storage class that is configured for local data store mode.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
scMode = getDataStore(cm,"mode","StorageClass");
```

Get Code Identifier Configured for Local Data Store

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the code identifier configured for the local data store mode.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
idDSMmode = getDataStore(cm,"mode","Identifier");
```

Get Calibration Properties Configured for Local Data Store

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the calibration properties such as calibration access and display identifier configured for local data store mode.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
prop_export = getDataStore(cm,"mode","Export");  
prop_bitmask = getDataStore(cm,"mode","BitMask");  
prop_calaccess = getDataStore(cm,"mode","CalibrationAccess");  
prop_compname = getDataStore(cm,"mode","CompuMethod");
```

```
prop_dispid = getDataStore(cm,"mode","DisplayIdentifier");
prop_format = getDataStore(cm,"mode","Format");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

dataStore — Block path, block handle, or name of data store

character vector | string scalar | block handle

Path of the Data Store Memory block for which to return the code mapping information, specified as a character vector or string scalar. Alternatively, you can specify a block handle or the name of the data store. If you specify the name of a data store and that name is not unique within the model, Simulink returns an error that instructs you to specify the block path or handle.

Example: `blockHandle`

Data Types: `char` | `string` | `block_handle`

property — Code mapping property value to return

`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name | `Export` | `BitMask` | `CalibrationAccess` | `CompuMethod` | `DisplayIdentifier` | `Format` | `MeasurementService`

Code mapping property for which to return a value. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	<code>StorageClass</code>
Name of variable for data store in the generated code	<code>Identifier</code>
Name of source definition file that contains definitions for global data that is read by the data store and external code	<code>DefinitionFile</code>
Name of <code>get</code> function called by code generated for the data store	<code>GetFunction</code>
Name of source header file that contains declarations for global data that is read by the data store and external code	<code>HeaderFile</code>
Name of memory section that contains data read by the state	<code>MemorySection</code>
Name of model for which the code generator places the definition for a data store shared by multiple models in a model hierarchy	<code>Owner</code>

Information to Return	Property Name
Boolean value indicating whether the code generator preserves dimensions of a data store that is represented as a multidimensional array	PerserveDimensions
Name of <code>set</code> function called by code generated for a data store	SetFunction
Name of structure in generated code for a data store	StructName
Boolean value indicating whether to export the selected data store to a calibration file (a2l)	Export
Mask value in hexadecimal format to extract single bits from the data store in a calibration tool	BitMask
Enumeration value indicating the access of calibration. <code>Calibration</code> for a data store indicates that the data store can be calibrated. <code>NoCalibration</code> indicates that the data store can be read-only and cannot be calibrated. <code>NoCalibration</code> is the default value for the property	CalibrationAccess
Name of the conversion method used during the calibration	CompuMethod
Optional display name of the data store for calibration	DisplayIdentifier
Specifies the display format of the data store being measured in a calibration tool	Format
Name of measurement service defined in Embedded Coder Dictionary	MeasurementService

Example: "StorageClass"

Example: "CalibrationAccess"

Output Arguments

propertyValue — Name of storage class or value of storage class property

character vector

Name of the storage class or value of the specified storage class property configured for the specified data store.

Data Types: `char`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `setDataDefault` | `setDataStore`

Topics

“Configure Data Stores for C Code Generation”

“C Data Code Interface Configuration for Model Interface Elements”

getInport

Get code and calibration configuration from code mappings for root-level inport

Syntax

```
propertyValue = getInport(myCodeMappingObj,inportBlock,property)
```

Description

`propertyValue = getInport(myCodeMappingObj,inportBlock,property)` returns the value of a code mapping property or calibration property for the specified root-level Inport block. For example, use this function to return the storage class or the value of a storage class property configured for a root-level inport in a model, or to return the calibration access of the inport.

Examples

Get Storage Class Configured for Root-Level Inport

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the name of the storage class that is configured for root-level inport `In1`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
scIn1 = getInport(cm,"In1","StorageClass");
```

Get Code Identifier Configured for Root-Level Inport

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the code identifier configured for root-level inport `In1`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
idIn1 = getInport(cm,"In1","Identifier");
```

Get Calibration Properties Configured for Root-Level Inport

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the calibration properties such as calibration access and display identifier configured for root-level inport `In1`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
prop_export = getInport(cm,"In1","Export");  
prop_bitmask = getInport(cm,"In1","BitMask");  
prop_calaccess = getInport(cm,"In1","CalibrationAccess");  
prop_compname = getInport(cm,"In1","CompuMethod");
```

```
prop_dispid = getInport(cm,"In1","DisplayIdentifier");
prop_format = getInport(cm,"In1","Format");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

inportBlock — Name, path, or handle of root-level inport

character vector | string scalar | block handle

Name, path, or handle of the root-level inport for which to return the code mapping information.

Example: `"In1"`

Data Types: `char` | `string` | `block_handle`

property — Code mapping property value to return

`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name | `Export` | `BitMask` | `CalibrationAccess` | `CompuMethod` | `DisplayIdentifier` | `Format` | `ReceiverService`

Code mapping property for which to return a value. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	<code>StorageClass</code>
Name of variable for inport in the generated code	<code>Identifier</code>
Name of source definition file that contains definitions for global data that is read by the inport and external code	<code>DefinitionFile</code>
Name of <code>get</code> function called by code generated for the inport	<code>GetFunction</code>
Name of source header file that contains declarations for global data that is read by the inport and external code	<code>HeaderFile</code>
Name of model for which the code generator places the definition for an inport shared by multiple models in a model hierarchy	<code>Owner</code>
Boolean value indicating whether the code generator preserves dimensions of an inport that is represented as a multidimensional array	<code>PreserveDimensions</code>
Name of <code>set</code> function called by code generated for an inport	<code>SetFunction</code>

Information to Return	Property Name
Name of structure in generated code for an inport	StructName
Boolean value indicating whether to export the selected inport to a calibration file (a2l)	Export
Mask value in hexadecimal format to extract single bits from the inport in a calibration tool	BitMask
Enumeration value indicating the access of calibration. Calibration for an inport indicates that the inport can be calibrated. NoCalibration indicates that the inport can be read-only but cannot be calibrated. NoCalibration is the default value for the property	CalibrationAccess
Name of the conversion method used during the calibration	CompuMethod
Optional display name of the inport for calibration	DisplayIdentifier
Specifies the display format of the inport being measured in a calibration tool	Format
Name of receiver service defined in Embedded Coder Dictionary	ReceiverService

Example: "StorageClass"

Example: "CalibrationAccess"

Output Arguments

propertyValue — Name or value of the property returned

character vector | boolean

Name or value of the property configured for the specified root-level inport.

Data Types: char | logical

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `setDataDefault` | `setInport`

Topics

"Configure Root-Level Inport Blocks for C Code Generation"

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Sender and Receiver Service Interfaces for Model Inports and Outports"

getModelParameter

Get code and calibration configuration from code mappings for model parameters

Syntax

```
propertyValue = getModelParameter(myCodeMappingObj,modelParameter,property)
```

Description

`propertyValue = getModelParameter(myCodeMappingObj,modelParameter,property)` returns the value of a code mapping property or calibration property for the specified model workspace parameter or model parameter argument. For example, use this function to return the storage class or the calibration access of a calibration property configured for the parameter or parameter argument.

Examples

Get Storage Class Configured for Model Parameter

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the name of the storage class that is configured for model parameter `K1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
scK1 = getModelParameter(cm,"K1","StorageClass");
```

Get Identifier Configured for Model Parameter

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, configure the storage class and identifier for model parameter `Table1`. To access the identifier configured for a model parameter, use the `getModelParameter` function.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setModelParameter(cm,"Table1",StorageClass="ExportedGlobal",Identifier="mp_Table1")
idTable1 = getModelParameter(cm,"Table1","Identifier")

idTable1 =

    'mp_Table1'
```

Get Storage Class Configured for Model Parameter Arguments

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the storage class configured for model parameter arguments `LOWER` and `UPPER`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
```

```
scLOWER = getModelParameter(cm,"LOWER","StorageClass");
scUPPER = getModelParameter(cm,"UPPER","StorageClass");
```

Get Calibration Properties Configured for Model Parameter

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the calibration properties such as calibration access and display identifier configured for model parameter `LOWER`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
prop_export = getModelParameter(cm,"LOWER","Export");
prop_bitmask = getModelParameter(cm,"LOWER","BitMask");
prop_calaccess = getModelParameter(cm,"LOWER","CalibrationAccess");
prop_compname = getModelParameter(cm,"LOWER","CompuMethod");
prop_dispid = getModelParameter(cm,"LOWER","DisplayIdentifier");
prop_format = getModelParameter(cm,"LOWER","Format");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

modelParameter — Name of model parameter or model parameter argument

character vector | string scalar

Name of the model workspace parameter or model parameter argument for which to return the code mapping information.

Example: `"Table1"`

Data Types: `char` | `string`

property — Code mapping property value to return

`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name | `Export` | `BitMask` | `CalibrationAccess` | `CompuMethod` | `DisplayIdentifier` | `Format` | `ParameterTuningService` | `ParameterArgumentTuningService`

Code mapping property for which to return a value. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	<code>StorageClass</code>
Name of variable for the parameter or parameter argument in the generated code	<code>Identifier</code>

Information to Return	Property Name
Name of source definition file that contains definitions for global data that is read by the parameter or parameter argument and external code	DefinitionFile
Name of <code>get</code> function called by code generated for the parameter or parameter argument	GetFunction
Name of source header file that contains declarations for global data that is read by the parameter or parameter argument and external code	HeaderFile
Name of model for which the code generator places the definition for the parameter or parameter argument shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether the code generator preserves dimensions of a parameter or parameter argument that is represented as a multidimensional array	PerserveDimensions
Name of <code>set</code> function called by code generated for the parameter or parameter argument	SetFunction
Name of structure in generated code for the parameter or parameter argument	StructName
Boolean value indicating whether to export the selected model parameter to a calibration file (a2l)	Export
Mask value in hexadecimal format to extract single bits from the model parameter in a calibration tool	BitMask
Enumeration value indicating the access of calibration. <code>Calibration</code> for an parameter indicates that the parameter can be calibrated and is the default value for the property. <code>NoCalibration</code> indicates that the parameter can be read-only and cannot be calibrated.	CalibrationAccess
Name of the conversion method used during the calibration	CompuMethod
Optional display name of the model parameter for calibration	DisplayIdentifier
Specifies the display format of the model parameter being calibrated in a calibration tool	Format
Name of parameter argument tuning service defined in Embedded Coder Dictionary	ParameterArgumentTuningService
Name of parameter tuning service defined in Embedded Coder Dictionary	ParameterTuningService

Example: "StorageClass"

Example: "CalibrationAccess"

Output Arguments

propertyValue — Name of storage class or value of storage class property

character vector

Name of the storage class or value of the specified storage class property configured for model parameter or parameter argument .

Data Types: char

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `setDataDefault` | `setModelParameter`

Topics

"Configure Parameters for C Code Generation"

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments"

getModelVariantControl

Get code configuration from code mappings for Simulink.VariantControl object

Syntax

```
propertyValue = getModelVariantControl(codeMappingObj,variantControl,property)
```

Description

propertyValue = getModelVariantControl(codeMappingObj,variantControl,property) returns the value of a code mapping property or calibration property for the specified Simulink.VariantControl object. Use this function to return the storage class or the calibration access of a calibration property configured for the variant control. Specifying code generation attributes for a Simulink.VariantControl object using code mappings is supported when the object is defined in the model workspace and its Value property is set to a Simulink.Parameter object.

Examples

Get Storage Class Configured for Simulink.VariantControl Object

From the model code mappings for the model slxVariantParameterCodeMappings, get the name of the storage class of variant control Wiper.

```
openExample("simulinkcoder/ConfigureCodeGenForVariantParametersInModelWorkspaceExample");
cm = coder.mapping.api.get("slxVariantParameterCodeMappings");
getModelVariantControl(cm,"Wiper","StorageClass");
```

Input Arguments

codeMappingObj — Code mapping

CodeMapping object

Code mapping object (model code mappings) returned by a call to function coder.mapping.api.get.

variantControl — Name of Simulink.VariantControl object

character vector | string scalar

Name of Simulink.VariantControl object specified as a character vector or string scalar.

Example: "varCtrl"

Data Types: char | string

property — Code mapping property name

StorageClass | Identifier | DefinitionFile | GetFunction | HeaderFile | Owner | PreserveDimensions | SetFunction | StructName | storage class property name | Export |

BitMask | CalibrationAccess | CompuMethod | DisplayIdentifier | Format |
ParameterTuningService | ParameterArgumentTuningService

Code mapping property name. The `getModelVariantControl` function returns the value of this property. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for the variant control in the generated code	Identifier
Name of source definition file that contains definitions for global data that is read by the variant control and external code	DefinitionFile
Name of <code>get</code> function called by code generated for the variant control	GetFunction
Name of source header file that contains declarations for global data that is read by the variant control and external code	HeaderFile
Name of model for which the code generator places the definition for the variant control shared by multiple models in a model hierarchy	Owner
Name of <code>set</code> function called by code generated for the variant control	SetFunction
Name of structure in generated code for the variant control	StructName
Boolean value indicating whether the code generator preserves dimensions of a variant control that is represented as a multidimensional array	PerserveDimensions
Name of <code>set</code> function called by code generated for the variant control	SetFunction
Name of structure in generated code for the variant control	StructName
Boolean value indicating whether to export the selected variant control to an A2L calibration file	Export
Mask value in hexadecimal format to extract single bits from the variant control in a calibration tool	BitMask

Information to Return	Property Name
Enumeration value indicating the access of calibration: • Calibration for a variant control indicates that the variant control can be calibrated and is the default value for the property. • NoCalibration indicates that the variant control can be read-only and cannot be calibrated.	CalibrationAccess
Name of the conversion method used during the calibration	CompuMethod
Optional display name of the variant control for calibration	DisplayIdentifier
Display format of the variant control being calibrated in a calibration tool	Format
Name of parameter argument tuning service defined in Embedded Coder Dictionary	ParameterArgumentTuningService
Name of parameter tuning service defined in Embedded Coder Dictionary	ParameterTuningService

Example: "StorageClass"

Output Arguments

propertyValue — Name of storage class or value of storage class property

character vector

Name of the storage class or value of the specified storage class property configured for the variant control, returned as a character vector.

Version History

Introduced in R2025a

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `setModelVariantControl` | `find` | `Simulink.VariantControl`

Topics

"Configure Code Generation for Variant Parameters in Model Workspace Using Code Mappings Editor"

"Generate Code for Instance-Specific Variation of Variant Parameter Values in Model Reference Hierarchy"

"Configure Parameters for C Code Generation"

"Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments"

getModelVariantVariable

Get code configuration from code mappings for `Simulink.VariantVariable` object

Syntax

```
propertyValue = getModelVariantVariable(codeMappingObj,variantVariable,  
property)
```

Description

`propertyValue = getModelVariantVariable(codeMappingObj,variantVariable,property)` returns the value of a code mapping property for the specified `Simulink.VariantVariable` object `variantVariable` (also referred to as a variant parameter) from the code mapping `codeMappingObj`. A `Simulink.VariantVariable` object derives its code mappings from the `Simulink.Parameter` object that is used to set the `Specification` property of the object. The `getModelVariantVariable` function returns the storage class and storage class property settings configured for the `Simulink.VariantVariable` object from this `Simulink.Parameter` object. For code mappings support, the `Simulink.VariantVariable` object and its associated `Simulink.VariantControl` and `Simulink.Parameter` objects must be defined in the model workspace of the model.

Examples

Get Storage Class Configured for `Simulink.VariantVariable` object

From the model code mappings for the model `slexVariantParameterCodeMappings`, get the name of the storage class of the variant variable `GearRatio`.

```
openExample("simulinkcoder/ConfigureCodeGenForVariantParametersInModelWorkspaceExample");  
cm = coder.mapping.api.get("slexVariantParameterCodeMappings");  
getModelVariantVariable(cm,"GearRatio","StorageClass");
```

Input Arguments

codeMappingObj — Code mapping

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

variantVariable — Name of `Simulink.VariantVariable` object

character vector | string scalar

Name of `Simulink.VariantVariable` object, specified as a character vector or string scalar.

Example: "GearRatio"

Data Types: char | string

property — Code mapping property name

StorageClass | Identifier | DefinitionFile | GetFunction | HeaderFile | Owner |
 PreserveDimensions | SetFunction | StructName | storage class property name |
 ParameterTuningService | ParameterArgumentTuningService

Code mapping property name. The `getModelVariantVariable` function returns the value of this property. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for the variant parameter in the generated code	Identifier
Name of source definition file that contains definitions for global data that is read by the variant parameter and external code	DefinitionFile
Name of <code>get</code> function called by code generated for the variant parameter	GetFunction
Name of source header file that contains declarations for global data that is read by the variant parameter and external code	HeaderFile
Name of model for which the code generator places the definition for the variant parameter shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether the code generator preserves dimensions of a variant parameter that is represented as a multidimensional array	PreserveDimensions
Name of <code>set</code> function called by code generated for the variant parameter	SetFunction
Name of structure in generated code for the variant parameter	StructName
Name of parameter argument tuning service defined in Embedded Coder Dictionary	ParameterArgumentTuningService
Name of parameter tuning service defined in Embedded Coder Dictionary	ParameterTuningService

Example: "StorageClass"

Output Arguments**propertyValue — Name of storage class or value of storage class property**

character vector

Name of the storage class or value of the specified storage class property configured for the `Simulink.VariantVariable` object, returned as a character vector.

Version History

Introduced in R2025a

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `setModelVariantControl` | `find` | `Simulink.VariantVariable`

Topics

"Configure Code Generation for Variant Parameters in Model Workspace Using Code Mappings Editor"

"Generate Code for Instance-Specific Variation of Variant Parameter Values in Model Reference Hierarchy"

"Configure Parameters for C Code Generation"

"Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments"

getOutput

Get code and calibration configuration from code mappings for root-level output

Syntax

```
propertyValue = getOutput(myCodeMappingObj, outputBlock, property)
```

Description

`propertyValue = getOutput(myCodeMappingObj, outputBlock, property)` returns the value of a code mapping property or calibration property for the specified root-level Output block. For example, use this function to return the storage class or the value of a storage class property configured for a root-level output in a model, or to return the calibration access of the output.

Examples

Get Storage Class Configured for Root-Level Output

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the name of the storage class that is configured for root-level output `Out1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
scOut1 = getOutput(cm,"Out1","StorageClass");
```

Get Code Identifier Configured for Root-Level Output

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the code identifier configured for root-level output `Out1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
idOut1 = getOutput(cm,"Out1","Identifier");
```

Get Calibration Properties Configured for Root-Level Output

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the calibration properties such as calibration access and display identifier configured for root-level output `Out1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
prop_export = getOutput(cm,"Out1","Export");
prop_bitmask = getOutput(cm,"Out1","BitMask");
prop_calaccess = getOutput(cm,"Out1","CalibrationAccess");
prop_compname = getOutput(cm,"Out1","CompuMethod");
```

```
prop_dispid = getOutputport(cm,"Out1","DisplayIdentifier");
prop_format = getOutputport(cm,"Out1","Format");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

outportBlock — Name, path, or handle of root-level outport

character vector | string scalar | block handle

Name, path, or handle of the root-level outport for which to return the code mapping information.

Example: `"Out1"`

Data Types: `char` | `string`

property — Code mapping property value to return

`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name | `Export` | `BitMask` | `CalibrationAccess` | `CompuMethod` | `DisplayIdentifier` | `Format` | `SenderService`

Code mapping property for which to return a value. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	<code>StorageClass</code>
Name of variable for root-level outport in the generated code	<code>Identifier</code>
Name of source definition file that contains definitions for global data that is read by the root-level outport and external code	<code>DefinitionFile</code>
Name of <code>get</code> function called by code generated for the root-level outport	<code>GetFunction</code>
Name of source header file that contains declarations for global data that is read by the root-level outport and external code	<code>HeaderFile</code>
Name of model for which the code generator places the definition for root-level outport shared by multiple models in a model hierarchy	<code>Owner</code>
Boolean value indicating whether the code generator preserves dimensions of a root-level outport that is represented as a multidimensional array	<code>PerserveDimensions</code>

Information to Return	Property Name
Name of set function called by code generated for root-level output	SetFunction
Name of structure in generated code for root-level output	StructName
Boolean value indicating whether to export the selected output to a calibration file (a2l)	Export
Mask value in hexadecimal format to extract single bits from the output in a calibration tool	BitMask
Enumeration value indicating the access of calibration. Calibration for an output indicates that the output can be calibrated. NoCalibration indicates that the output can be read-only but cannot be calibrated. NoCalibration is the default value for the property	CalibrationAccess
Name of the conversion method used during the calibration	CompuMethod
Optional display name of the output for calibration	DisplayIdentifier
Specifies the display format of the output being measured in a calibration tool	Format
Name of sender service defined in Embedded Coder Dictionary	SenderService

Example: "StorageClass"

Example: "CalibrationAccess"

Output Arguments

propertyValue — Name or value of the property returned

character vector | boolean

Name or value of the property configured for the specified root-level output.

Data Types: char | logical

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `setDataDefault` | `setOutput`

Topics

"Configure Root-Level Output Blocks for C Code Generation"

“C Data Code Interface Configuration for Model Interface Elements”

getSignal

Get code and calibration configuration from code mappings for block output signal

Syntax

```
propertyValue = getSignal(myCodeMappingObj,portHandle,property)
```

Description

`propertyValue = getSignal(myCodeMappingObj,portHandle,property)` returns the value of a code mapping property or calibration property for the signal specified by a block output port handle. For example, use this function to return the name of the storage class or calibration access of a measurement property configured for a signal.

This function does not apply to signals that originate from root-level Inport blocks. For signals that originate from root-level Inport blocks, see `getInport`.

Examples

Get Storage Class Configured for Block Output Signal

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the name of the storage class that is configured for the output signal of lookup table block `Table1`. After creating the object `cm` by calling function `coder.mapping.api.get`, get the handle to the output signals for the lookup table block. Get the storage class configured for the output port by calling `getSignal`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
lut1_ports = get_param("ConfigurationRapidPrototypingInterface/Table1","PortHandles");
lut1_outPort = lut1_ports.Outport;
addSignal(cm, lut1_outPort);
scTable1D = getSignal(cm,lut1_outPort,"StorageClass");
```

Get Code Identifiers Configured for Block Output Signals

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the code identifiers that are configured for output signals of lookup table blocks `Table1` and `Table2`. After creating the object `cm` by calling function `coder.mapping.api.get`, get the handles to the output ports for the lookup table blocks. Get the code identifiers configured for the output ports by calling `getSignal`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
lut1_ports = get_param("ConfigurationRapidPrototypingInterface/Table1","PortHandles");
lut2_ports = get_param("ConfigurationRapidPrototypingInterface/Table2","PortHandles");
lut1_outPort = lut1_ports.Outport;
lut2_outPort = lut2_ports.Outport;
addSignal(cm, lut1_outPort);
addSignal(cm, lut2_outPort);
setSignal(cm,lut1_outPort,"StorageClass","ExportedGlobal")
```

```
setSignal(cm, lut2_outPort, "StorageClass", "ExportedGlobal")
idTable1 = getSignal(cm, lut1_outPort, "Identifier");
idTable2 = getSignal(cm, lut2_outPort, "Identifier");
```

Get Calibration Properties Configured for Signal

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the calibration properties configured for signal `Signal_1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
port_handles = get_param("ConfigurationRapidPrototypingInterface/Table1", "PortHandles");
Signal_1 = port_handles.Outport;
addSignal(cm, Signal_1);
prop_export = getSignal(cm, Signal_1, "Export");
prop_bitmask = getSignal(cm, Signal_1, "BitMask");
prop_calaccess = getSignal(cm, Signal_1, "CalibrationAccess");
prop_compname = getSignal(cm, Signal_1, "CompuMethod");
prop_dispid = getSignal(cm, Signal_1, "DisplayIdentifier");
prop_format = getSignal(cm, Signal_1, "Format");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

portHandle — Output port handle of signal source block

port handle

Block output signals for which to return signal code mapping information.

Example: `portHandle`

Data Types: `port_handle`

property — Code mapping property value to return

`StorageClass` | `Identifier` | `DefinitionFile` | `GetFunction` | `HeaderFile` | `Owner` | `PreserveDimensions` | `SetFunction` | `StructName` | storage class property name | `Export` | `BitMask` | `CalibrationAccess` | `CompuMethod` | `DisplayIdentifier` | `Format` | `MeasurementService`

Code mapping property for which to return a value. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	<code>StorageClass</code>
Name of variable for signal data in the generated code	<code>Identifier</code>

Information to Return	Property Name
Name of source definition file that contains definitions for global data that is read by the signal data and external code	DefinitionFile
Name of <code>get</code> function called by code generated for the signal data	GetFunction
Name of source header file that contains declarations for global data that is read by the signal data and external code	HeaderFile
Name of model for which the code generator places the definition for a signal data shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether the code generator preserves dimensions of a signal data that is represented as a multidimensional array	PerserveDimensions
Name of <code>set</code> function called by code generated for a signal data	SetFunction
Name of structure in generated code for a signal data	StructName
Boolean value indicating whether to export the selected signal to a calibration file (a2l)	Export
Mask value in hexadecimal format to extract single bits from the signal in a calibration tool	BitMask
Enumeration value indicating the access of calibration. <code>Calibration</code> for a signal indicates that the signal can be calibrated. <code>NoCalibration</code> indicates that the signal can be read-only but cannot be calibrated. <code>NoCalibration</code> is the default value for the property	CalibrationAccess
Name of the conversion method used during the calibration	CompuMethod
Optional display name of the signal for calibration	DisplayIdentifier
Specifies the display format of the signal being measured in a calibration tool	Format
Name of measurement service defined in Embedded Coder Dictionary	MeasurementService

Example: "StorageClass"

Example: "CalibrationAccess"

Output Arguments

propertyValue — Name of storage class or value of storage class property
character vector

Name of the storage class or value of the specified storage class property configured for the specified signal.

Data Types: `char`

Version History

Introduced in R2020b

See Also

`addSignal` | `coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `removeSignal` | `setDataDefault` | `setSignal`

Topics

“Configure Signal Data for C Code Generation”

“C Data Code Interface Configuration for Model Interface Elements”

“Configure Measurement Service Interfaces for Signals, States, and Data Stores”

getState

Get code and calibration configuration from code mappings for block state

Syntax

```
propertyValue = getState(myCodeMappingObj,block,property)
```

Description

`propertyValue = getState(myCodeMappingObj,block,property)` returns the value of a code mapping property or calibration property for the state of the specified block. For example, use this function to return the storage class or calibration access of a measurement property configured for a block state.

Examples

Get Storage Class Configured for Block State

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the name of the storage class that is configured for state X of Unit Delay block Delay.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
state = find(cm,"State");
scX = getState(cm,state,"StorageClass");
```

Get Calibration Properties Configured for State of Unit Delay Block

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, get the calibration properties configured for state X of Unit Delay block Delay.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
state = find(cm,"State");
prop_export = getState(cm,state,"Export");
prop_bitmask = getState(cm,state,"BitMask");
prop_calaccess = getState(cm,state,"CalibrationAccess");
prop_compname = getState(cm,state,"CompuMethod");
prop_dispid = getState(cm,state,"DisplayIdentifier");
prop_format = getState(cm,state,"Format");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

block — Path or handle of block

character vector | string scalar | block handle

Path of the block for which to return the state code mapping information, specified as a character vector or string scalar. Alternatively, you can specify a block handle.

Example: blockHandle

Data Types: char | string | block_handle

property — Code mapping property value to return

StorageClass | Identifier | Export | BitMask | CalibrationAccess | DisplayIdentifier | Format | CompuMethod | ...

Code mapping property for which to return a value. For a storage class defined in the Embedded Coder Dictionary associated with the model, specify a property name or one of these property names.

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for state in the generated code (unavailable when storage class is specified as Auto)	Identifier
Name of source definition file that contains definitions for global data that is read by the state and external code	DefinitionFile
Name of <code>get</code> function called by code generated for the state	GetFunction
Name of source header file that contains declarations for global data that is read by the state and external code	HeaderFile
Name of memory section that contains data read by the state	MemorySection
Name of model for which the code generator places the definition for a state shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether code generator preserves dimensions of a state that is represented as a multidimensional array	PerserveDimensions
Name of <code>set</code> function called by code generated for a state	SetFunction
Name of structure in generated code for a state	StructName
Boolean value indicating whether to export the selected state to a calibration file (a2l)	Export
Mask value in hexadecimal format to extract single bits from the state in a calibration tool	BitMask

Information to Return	Property Name
Enumeration value indicating the access of calibration. Calibration for a state indicates that the state can be calibrated. NoCalibration indicates that the state can be read-only but cannot be calibrated. NoCalibration is the default value for the property	CalibrationAccess
Name of the conversion method used during the calibration	CompuMethod
Optional display name of the state for calibration	DisplayIdentifier
Specifies the display format of the state being measured in a calibration tool	Format
Name of the measurement service	MeasurementService

Example: "StorageClass"

Example: "CalibrationAccess"

Output Arguments

propertyValue — Name of storage class or value of storage class property

character vector

Name of the storage class or value of the specified storage class property configured for the specified block state, returned as a character vector.

Data Types: char

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `setDataDefault` | `setState`

Topics

"Configure Block States for C Code Generation"

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Measurement Service Interfaces for Signals, States, and Data Stores"

removeSignal

Remove block output signal from model code mappings

Syntax

```
removeSignal(coderMapObj,portHandle)
```

Description

`removeSignal(coderMapObj,portHandle)` removes signals specified by the block output port handles from the specified model code mappings.

This function does not apply to signals that originate from root-level Inport blocks.

Examples

Programmatically Add and Remove Signal Lines in Code Mappings of Simulink® Models

Use the programmatic interface to add and remove signals in the code mapping configuration of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure it is open with the **Signals/States** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model `CoderMapAPI`.

```
simulinkModel = "CoderMapAPI";  
open_system(simulinkModel);
```

Use the function `coder.mapping.api.get` to retrieve the code mapping object of the model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Get the handles of the output ports of the four Gain blocks and store them in variables.

```
port_handle_1 = get_param(simulinkModel+"/gain_1","PortHandles").Output;  
port_handle_2 = get_param(simulinkModel+"/gain_2","PortHandles").Output;  
port_handle_3 = get_param(simulinkModel+"/gain_3","PortHandles").Output;  
port_handle_4 = get_param(simulinkModel+"/gain_4","PortHandles").Output
```

```
port_handle_4 =  
305.0021
```

Use the function `addSignal` to add the signal to the code mappings of the model. Set the storage class of `sig_1` and `sig_3` to `ImportedExtern` and the storage class of `sig_2` and `sig_4` to `ImportedExternPointer`. Update the model.

```
addSignal(codeMapObj,[port_handle_1,port_handle_3],StorageClass="ImportedExtern")  
addSignal(codeMapObj,[port_handle_2,port_handle_4],StorageClass="ImportedExternPointer")  
set_param(simulinkModel,"SimulationCommand","update")
```

Use the `slbuild` command to generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Signals with `ImportedExtern` and `ImportedExternPointer` storage classes are declared in separate sections of the generated private header file of the model. Store the name of the private header file in the variable `priv_h_file`.

```
priv_h_file = fullfile(simulinkModel+"_grt_rtw",simulinkModel+"_private.h")

priv_h_file =
"CoderMapAPI_grt_rtw\CoderMapAPI_private.h"
```

The declarations of the signals appear as the following code in the header file:

```
/* Imported (extern) block signals */
extern real_T sig_1;          /* '<Root>/gain_1' */
extern real_T sig_3;          /* '<Root>/gain_3' */
extern real_T in_port_1;      /* '<Root>/in_port_1' */
extern real_T in_port_2;      /* '<Root>/in_port_2' */

/* Imported (extern) pointer block signals */
extern real_T *sig_2;          /* '<Root>/gain_2' */
extern real_T *sig_4;          /* '<Root>/gain_4' */
extern real_T *in_port_3;      /* '<Root>/in_port_3' */
extern real_T *in_port_4;      /* '<Root>/in_port_4' */
```

The signals are defined according to the specified storage classes.

To open the header file, enter this command in the MATLAB® Command Window.

```
edit(priv_h_file)
```

Use the function `removeSignal` to remove the first two signal lines from the code mappings of the model. Update the model.

```
removeSignal(codeMapObj,[port_handle_1,port_handle_2])
set_param(simulinkModel,"SimulationCommand","update")
```

Rebuild the model with the `slbuild` command.

```
evalc("slbuild(simulinkModel)");
```

The updated sections appear as the following code in the header file.

```
/* Imported (extern) block signals */
extern real_T sig_3;          /* '<Root>/gain_3' */
extern real_T in_port_1;      /* '<Root>/in_port_1' */
extern real_T in_port_2;      /* '<Root>/in_port_2' */

/* Imported (extern) pointer block signals */
extern real_T *sig_4;          /* '<Root>/gain_4' */
extern real_T *in_port_3;      /* '<Root>/in_port_3' */
extern real_T *in_port_4;      /* '<Root>/in_port_4' */
```

The removed signals no longer appear in the header file.

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

portHandle — Output port handles of signal source blocks

block output port handle | port handle array

Signal to remove from the code mappings, specified as the output port handle of the source block of the signal. To remove multiple signals simultaneously, specify an output port handle array.

Example: `removeSignal(coderMapObj,[outport_1_Handle,outport_2_Handle])`

Version History

Introduced in R2020b

See Also

`addSignal` | `coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getSignal` | `setDataDefault` | `setSignal`

Topics

“Configure Signal Data for C Code Generation”

“C Data Code Interface Configuration for Model Interface Elements”

setDataDefault

Set default storage class and storage class property values for model data category

Syntax

```
setDataDefault(coderMapObj,elemCategory,Name=Value)
```

Description

`setDataDefault(coderMapObj,elemCategory,Name=Value)` sets the default storage class and storage class property values in the code mappings for the specified category of model data.

You cannot specify default data interfaces for models with an attached Embedded Coder Dictionary that defines a service interface configuration.

Examples

Programmatically Get and Set Data Defaults in Code Mappings of Simulink® Models

Use the programmatic interface to get and set the data defaults in the code mappings configuration of a Simulink model.

To interactively observe how your commands are reflected in the **Code Mappings** editor, make sure the **Code Mappings** editor is open with the **Data Defaults** tab selected. To learn how to open the **Code Mappings** editor, see “Open the Code Mappings Editor — C” on page 22-0 .

Open the model `CoderMapAPI`.

```
simulinkModel = "CoderMapAPI";
open_system(simulinkModel);
```

Retrieve the code mappings object of the model.

```
codeMapObj = coder.mapping.api.get(simulinkModel);
```

Set the storage class of all root-level ports to `Model default` so that their generated code is determined by the value you set for `Data Defaults`.

```
setInport(codeMapObj,find(codeMapObj,"Inport"),StorageClass="Model default")
setOutport(codeMapObj,find(codeMapObj,"Outport"),StorageClass="Model default")
```

Determine the default storage classes for Input and output ports.

```
inPortDefaultStorageClass = getDataDefault(codeMapObj,"Inports","StorageClass")
```

```
inPortDefaultStorageClass =
'ImportedExtern'
```

```
outPortDefaultStorageClass = getDataDefault(codeMapObj,"Outports","StorageClass")
```

```
outPortDefaultStorageClass =
'ImportedExternPointer'
```

Generate code from the model.

```
evalc("slbuild(simulinkModel)");
```

Root-level ports with `ImportedExtern` and `ImportedExternPointer` storage classes are declared in the generated private header file of the model in separate sections.

Store the name of the header file.

```
priv_h_file = fullfile(simulinkModel+"_grt_rtw",simulinkModel+"_private.h")
```

```
priv_h_file =  
"CoderMapAPI_grt_rtw\CoderMapAPI_private.h"
```

These are the declarations of the root-level ports in the header file:

```
/* Data with Imported storage */  
extern real_T in_port_1;          /* '<Root>/in_port_1' */  
extern real_T in_port_2;          /* '<Root>/in_port_2' */  
extern real_T in_port_3;          /* '<Root>/in_port_3' */  
extern real_T in_port_4;          /* '<Root>/in_port_4' */  
  
/* Data with Imported storage (pointer) */  
extern real_T *out_port_1;        /* '<Root>/out_port_1' */  
extern real_T *out_port_2;        /* '<Root>/out_port_2' */  
extern real_T *out_port_3;        /* '<Root>/out_port_3' */  
extern real_T *out_port_4;        /* '<Root>/out_port_4' */
```

To open the header file, enter this command in the MATLAB® Command Window.

```
edit(priv_h_file)
```

Switch between the default storage class of input and output ports.

```
setDataDefault(codeMapObj,"Inports",StorageClass=outPortDefaultStorageClass);  
setDataDefault(codeMapObj,"Outports",StorageClass=inPortDefaultStorageClass);
```

Generate the code from the revised model.

```
evalc("slbuild(simulinkModel)");
```

The declarations of the root-level ports are updated in the header file according to the specified storage classes.

```
/* Data with Imported storage */  
extern real_T out_port_1;          /* '<Root>/out_port_1' */  
extern real_T out_port_2;          /* '<Root>/out_port_2' */  
extern real_T out_port_3;          /* '<Root>/out_port_3' */  
extern real_T out_port_4;          /* '<Root>/out_port_4' */  
  
/* Data with Imported storage (pointer) */  
extern real_T *in_port_1;          /* '<Root>/in_port_1' */  
extern real_T *in_port_2;          /* '<Root>/in_port_2' */  
extern real_T *in_port_3;          /* '<Root>/in_port_3' */  
extern real_T *in_port_4;          /* '<Root>/in_port_4' */
```

Input Arguments

coderMapObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

elemCategory — Model element category

Constants | ExternalParameterObjects | GlobalDataStores | Inports | InternalData | ModelParameters | ModelParameterArguments | Outports | SharedLocalDataStores

Category of model data element for which to set the storage class and storage class properties.

Example: `"Inports"`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

Auto | Bitfield | CompileFlag | Const | ConstVolatile | Define | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedDefine | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified data element category. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. Values that you can specify vary depending on the category that you specify. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

Example: `StorageClass="ImportedExtern"`

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `Const`, `ConstVolatile`, `ExportToFile`, and `Volatile`.

Example: `DefinitionFile="myDataDefs.c"`

Data Types: `char` | `string`

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Example: `GefFunction="myDataGetFunction"`

Data Types: char | string

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, Define, ExportToFile, GetSet, ImportedDefine, ImportFromFile, and Volatile.

Example: HeaderFile="myDataDecl.h"

Data Types: char | string

MemorySection — Name of memory section

character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Note You cannot directly retrieve or configure the default MemorySection for the ModelParameterArguments category using the code mapping object. To configure the default MemorySection for ModelParameterArguments, define a custom StorageClass with the MemorySection in a coder dictionary, and set this storage class as the default for ModelParameterArguments.

Example: MemorySection="myFastMem"

Data Types: char | string

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Example: Owner="myModelA"

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes Const, ConstVolatile, , ExportToFile, FileScope, GetSet, ImportFromFile, Localizable, and Volatile.

Example: PreserveDimensions=true

Data Types: logical

SetFunction — Name of set function

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Example: `SetFunction="myDataSetFunction"`

Data Types: `char` | `string`

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Example: `StructName="myDataStruct"`

storageClassPropertyName — Value of storage class property

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getDataDefault`

Topics

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Root-Level Inport Blocks for C Code Generation"

"Configure Root-Level Outport Blocks for C Code Generation"

"Configure Signal Data for C Code Generation"

"Configure Parameters for C Code Generation"

"Configure Block States for C Code Generation"

"Configure Data Stores for C Code Generation"

"Configure Default C Code Generation for Categories of Data Elements and Functions"

setDataStore

Configure local or shared local data store for code and calibration file (a2l) generation

Syntax

```
setDataStore(myCodeMappingObj,dataStore,Name,Value)
```

Description

`setDataStore(myCodeMappingObj,dataStore,Name,Value)` configures the specified local or shared local data store for code and calibration file (a2l) generation. Use this function to map a local or shared local data store to the storage class and storage class property settings that the code generator uses to produce C code. Or, set the calibration access for that data store.

Examples

Configure Storage Class for Local Data Store

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for local data store mode to `ExportedGlobal`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
setDataStore(cm,"mode",StorageClass="ExportedGlobal");
```

Configure Storage Class for Local and Shared Local Data Stores in Model to Model default

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for local and shared local data stores throughout the model to `Model default`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
dsmHandles = find(cm,"DataStores")  
setDataStore(cm,dsmHandles,StorageClass="Model default");
```

Configure Code Identifier for Local Data Store

In the model code mappings, for model `ConfigurationRapidPrototypingInterface`, set the code identifier for local data store mode to `ds_mode`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
setDataStore(cm,"mode",Identifier="ds_mode");
```

Configure Calibration Properties for Local Data Store

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the calibration properties configured for the local data store mode.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setDataStore(cm,"mode",Export=true);
setDataStore(cm,"mode",CalibrationAccess="Calibration");
setDataStore(cm,"mode",CompuMethod="CM_state");
setDataStore(cm,"mode",DisplayIdentifier="state_X");
setDataStore(cm,"mode",Format="%4.2");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

dataStore — Block path, block handle, or name of data store

character vector | string scalar | block handle | array of character vectors | array of string scalars | array of block handles

Path of the Data Store Memory block for which to return the code mapping information, specified as a character vector or string scalar. Alternatively, you can specify a block handle or the name of the data store. If you specify the name of a data store and that name is not unique within the model, Simulink returns an error that instructs you to specify the block path or handle. To specify multiple data stores, use an array.

Example: `blockHandle`

Data Types: `char` | `string` | `block_handle` | `array`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `StorageClass="ExportedGlobal`

StorageClass — Name of storage class

`Auto` | `Bitfield` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `FileScope` | `GetSet` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified data store. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Data Stores for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the local data store in the generated code.

Data Types: char | string

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: char | string

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: char | string

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: char | string

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data that is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

True | False

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: logical

SetFunction — Name of set function

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: char | string

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes **Bitfield** and **Struct**.

Data Types: char | string

storageClassName — Value of storage class property

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Export — Enable or disable generation of the data store to calibration file (a2l)

true (default) | false

Enable this property to generate the data store to calibration file (a2l).

BitMask — Extract single bits from the data store value

hexadecimal value

Mask value in hexadecimal format to extract single bits from the data store during calibration. This property is applicable only for integers.

Data Types: hexadecimal

CalibrationAccess — Enable or disable calibration of the data store

NoCalibration (default) | Calibration

Select **Calibration** for the data store to enable the calibration. Select **NoCalibration** to view the value of the data store and disable the calibration.

CompuMethod — Name of conversion method

character vector | string scalar

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Data Types: char | string

DisplayIdentifier — Display name of the data store

character vector | string scalar

Optional display name of the data store for the measurement purpose in the calibration tool, which is different than the data store name in the Simulink model.

Data Types: char | string

Format — Display format of the data store value

%[length].[layout]

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in **CompuMethod** of the data store.

MeasurementService — Name of measurement service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a measurement service interface defined in the Embedded Coder Dictionary. By configuring the measurement service interface for signals, states, and data stores, you can preserve the data in the generated code for measurement. To use the dictionary default, specify "Dictionary default". If data from the state does not need to be preserved in the code, specify "Not measured".

To configure the measurement service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see "Configure Measurement Service Interfaces for Signals, States, and Data Stores".

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getDataStore` | `setDataDefault`

Topics

"Configure Data Stores for C Code Generation"

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Measurement Service Interfaces for Signals, States, and Data Stores"

setInport

Configure root-level inports for code and calibration file (a2l) generation

Syntax

```
setInport(myCodeMappingObj,inport,Name,Value)
```

Description

`setInport(myCodeMappingObj,inport,Name,Value)` configures specified root-level Inport blocks for code and calibration file (a2l) generation. For example, use this function to map specified root-level inports to the storage class and storage class property settings that the code generator uses to produce C code for the inports. Or, set the calibration access for the inport.

Examples

Configure Storage Class for Root-Level Inports in Model to Model default

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for root-level inports throughout the model to `Model default`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
inBlockHandles = find(cm,"Inports");
setInport(cm,inBlockHandles,StorageClass="Model default");
```

Configure Storage Class and Storage Class Properties for Root Inport Block

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for root-level inport `In1` to `ExportedGlobal` and set the code identifier to `input1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setInport(cm,"In1",StorageClass="ExportedGlobal",Identifier="input1");
```

Configure Calibration Properties for Root-Level Inports

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the calibration properties for root-level Inport block `In1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setInport(cm,"In1",Export=true);
setInport(cm,"In1",CalibrationAccess="Calibration");
setInport(cm,"In1",CompuMethod="CM_Inport");
```

```
setInport(cm,"In1",DisplayIdentifier="inport_display_name");  
setInport(cm,"In1",Format="%4.2");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

inport — Name, path, or handle of root-level inport

character vector | string scalar | block handle | cell array of character vectors | cell array of string scalars | cell array of handles

Name, path, or handle of root-level inport to configure. To specify multiple inports, use a cell array.

Example: `"In1"`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

`Auto` | `Bitfield` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `GetSet` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified root Inport block. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Root-Level Inport Blocks for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the inport in the generated code.

Data Types: `char` | `string`

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: char | string

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: char | string

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

True | False

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: logical

SetFunction — Name of set function

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: char | string

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: char | string

storageClassPropertyName — Value of storage class property

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Export — Enable or disable generation of the inport to calibration file (a2l)`true (default) | false`

Enable this property to generate the inport to calibration file (a2l).

BitMask — Extract single bits from the inport value`hexadecimal value`

Mask value in hexadecimal format to extract single bits from the inport during calibration. This property is applicable only for integers.

Data Types: `hexadecimal`

CalibrationAccess — Enable or disable calibration of the inport`NoCalibration (default) | Calibration`

Select `Calibration` for the inport to enable the calibration. Select `NoCalibration` to view the value of the inport and disable the calibration.

CompuMethod — Name of conversion method`character vector | string scalar`

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Data Types: `char` | `string`

DisplayIdentifier — Display name of the inport`character vector | string scalar`

Optional display name of the inport for the measurement purpose in the calibration tool, which is different than the inport name in the Simulink model.

Data Types: `char` | `string`

Format — Display format of the inport value`%[length].[layout]`

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in `CompuMethod` of the inport.

ReceiverService — Name of receiver service interface`Dictionary default (default) | character vector | string scalar`

String or character vector containing the name of a receiver service interface defined in the Embedded Coder Dictionary. Within a target environment, a component receives data from other components by calling the target platform receiver service. To use the dictionary default, specify `"Dictionary default"`.

To configure the receiver service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see “Configure Sender and Receiver Service Interfaces for Model Inports and Outports”.

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getInport` | `setDataDefault`

Topics

“Configure Root-Level Inport Blocks for C Code Generation”

“C Data Code Interface Configuration for Model Interface Elements”

“Configure Sender and Receiver Service Interfaces for Model Inports and Outports”

setModelParameter

Configure model parameter for code and calibration file (a2l) generation

Syntax

```
setModelParameter(myCodeMappingObj,modelParameter,Name,Value)
```

Description

`setModelParameter(myCodeMappingObj,modelParameter,Name,Value)` configures the specified model parameter or model parameter argument for code and calibration file (a2l) generation. Use this function to map the specified model parameter or model parameter argument to the storage class and storage class property settings that the code generator uses to produce C code. Or, set the calibration access for the parameter.

Examples

Configure Storage Class for Model Parameter

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for model parameter `K1` to `ExportedGlobal`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
setModelParameter(cm,"K1",StorageClass="ExportedGlobal");
```

Configure Storage Class for Model Parameters in Model to Model default

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for model parameters throughout the model to `Model default`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
paramHandles = find(cm,"ModelParameters")  
setModelParameter(cm,paramHandles,StorageClass="Model default");
```

Configure Storage Class for Model Parameters

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for model parameters `Table1` and `Table2` to `ExportedGlobal`.

```
openExample("ConfigurationRapidPrototypingInterface");  
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");  
setModelParameter(cm,"Table1",StorageClass="ExportedGlobal",Identifier="mp_Table1");  
setModelParameter(cm,"Table2",StorageClass="ExportedGlobal",Identifier="mp_Table2");
```


Configure Calibration Properties for Model Parameter

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the calibration properties configured for the model parameter argument `LOWER`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setModelParameter(cm,"LOWER",Export=true);
setModelParameter(cm,"LOWER",CalibrationAccess="Calibration");
setModelParameter(cm,"LOWER",CompuMethod="CM_param");
setModelParameter(cm,"LOWER",DisplayIdentifier="param_lower");
setModelParameter(cm,"LOWER",Format="%4.2");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

modelParameter — Name of model parameter or model parameter argument

character vector | string | cell array of character vectors | cell array of string scalars

Name of the model workspace parameter or model parameter argument to configure.

Example: `"Table1"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

`Auto` | `Bitfield` | `CompilerFlag` | `Const` | `ConstVolatile` | `Define` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `FileScope` | `GetSet` | `ImportedDefine` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified model parameter or model parameter argument. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See "Configure Parameters for C Code Generation".

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the model parameter or model parameter argument in the generated code.

Data Types: char | string

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: char | string

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: char | string

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: char | string

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

True | False

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element that is represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: logical

SetFunction — Name of set function

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: char | string

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`

storageClassPropertyName — Value of storage class property

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Note For an element associated with the custom storage class where **Type** is set to **Other**, create a custom attributes class for the custom storage class and associate the custom attributes class with a Boolean property, `SupportSILPIL` set to true. For more information, see “Interfaces That Produce Additional Generated Code”.

Export — Enable or disable generation of the model parameter to calibration file (a2l)

`true` (default) | `false`

Enable this property to generate the model parameter to calibration file (a2l).

BitMask — Extract single bits from the model parameter value

hexadecimal value

Mask value in hexadecimal format to extract single bits from the model parameter during calibration. This property is applicable only for integers.

Data Types: `hexadecimal`

CalibrationAccess — Enable or disable calibration of the model parameter

`Calibration` (default) | `NoCalibration`

Select `Calibration` for the model parameter to enable the calibration. Select `NoCalibration` to view the value of the model parameter and disable the calibration.

CompuMethod — Name of conversion method

character vector | `string scalar`

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Data Types: `char` | `string`

DisplayIdentifier — Display name of the model parameter

character vector | `string scalar`

Optional display name of the model parameter for measurement purpose in the calibration tool, which is different than the model parameter name in the Simulink model.

Data Types: `char` | `string`

Format — Display format of the model parameter value

`%[length].[layout]`

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in `CompuMethod` of the model parameter.

ParameterTuningService — Name of parameter tuning service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a parameter tuning service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify "Dictionary default".

To configure the parameter tuning service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see "Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments".

Data Types: char | string

ParameterArgumentTuningService — Name of parameter argument tuning service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a parameter argument tuning service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify "Dictionary default".

To configure the parameter argument tuning service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see "Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments".

Data Types: char | string

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getModelParameter` | `setDataDefault`

Topics

"Configure Parameters for C Code Generation"

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments"

setModelVariantControl

Configure Simulink.VariantControl object for code generation

Syntax

```
setModelVariantControl(codeMappingObj,variantControl,Name=Value)
```

Description

setModelVariantControl(codeMappingObj,variantControl,Name=Value) configures the specified Simulink.VariantControl object variantControl in the code mappings codeMappingObj for code generation. Specifying code generation attributes for a Simulink.VariantControl object using code mappings is supported when the object is defined in the model workspace and its Value property is set to a Simulink.Parameter object. Use this function to map the specified variant control to the storage class and storage class property settings that the code generator uses to produce C code.

Examples

Configure Storage Class for Simulink.VariantControl object in Model Workspace

Get the code mappings for model slxVariantParameterCodeMappings.

```
openExample("simulinkcoder/ConfigureCodeGenForVariantParametersInModelWorkspaceExample");
cm = coder.mapping.api.get("slxVariantParameterCodeMappings");
```

Set the storage class for the variant control named Wiper defined in the model workspace to ExportedGlobal.

```
setModelVariantControl(cm,"Wiper",StorageClass="ExportedGlobal");
```

Input Arguments

codeMappingObj — Code mapping

CodeMapping object

Code mapping object (model code mappings) returned by a call to function coder.mapping.api.get.

variantControl — Name of Simulink.VariantControl object

character vector | string scalar

Name of Simulink.VariantControl object, specified as a character vector or string scalar.

Example: "varCtrl"

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

`Auto` | `Bitfield` | `CompilerFlag` | `Const` | `ConstVolatile` | `Define` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `FileScope` | `GetSet` | `ImportedDefine` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified variant control. `StorageClass` is the name of a predefined storage class or a storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Parameters for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the variant control in the generated code, specified as a character vector or string scalar.

Data Types: `char` | `string`

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code, specified as a character vector or string scalar. This argument applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code, specified as a character vector or string scalar. This argument applies to the storage class `GetSet`.

Data Types: `char` | `string`

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code, specified as a character vector or string scalar. This argument applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: `char` | `string`

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy, specified as a character vector or string scalar. The code generated for the model that owns the data includes the global data definition. This argument applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

PreserveDimensions — Option to preserve dimensions of multidimensional arrays

`true` or `1` | `false` or `0`

When model configuration parameter **Array layout** is set to `Row-major`, option to preserve dimensions of a data element that is represented in generated code as a multidimensional array, specified as a numeric or logical `1` (`true`) or `0` (`false`). This argument applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`

SetFunction — Name of set function

`character string` | `string scalar`

Name of a `set` function that a data element calls in the generated code, specified as a character vector or string scalar. This argument applies to the storage class `GetSet`.

Data Types: `char` | `string`

StructName — Name of structure

`character vector` | `string scalar`

Name that the code generator uses to identify the structure for a data element in the generated code. This argument applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`

storageClassName — Value of storage class property

depends on property definition

Value of storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Note For an element associated with the custom storage class where **Type** is set to **Other**, create a custom attributes class for the custom storage class and associate the custom attributes class with a Boolean property, `SupportSILPIL` set to `true`. For more information, see “Interfaces That Produce Additional Generated Code”.

Export — Option to enable or disable generation of the model parameter to A2L calibration file

`true` or `1` (default) | `false` or `0`

Option to enable or disable generation of the model parameter to an A2L calibration file, specified as a numeric or logical `1` (`true`) or `0` (`false`).

BitMask — Extract single bits from the variant control value

`hexadecimal integer`

Mask value to extract single bits from the variant control during calibration, specified as integer in hexadecimal format. This property is applicable only for integers.

Data Types: hexadecimal

CalibrationAccess — Option to enable or disable calibration of the variant control

"Calibration" (default) | "NoCalibration"

Option to enable or disable calibration of the variant control, specified as one of these values:

- "Calibration" — Enable calibration.
- "NoCalibration" — Disable calibration.

CompuMethod — Name of conversion method

character vector | string scalar

Name of the method for converting the ECU-internal value to a physical value for easy readability, specified as a character vector or string scalar.

Data Types: char | string

DisplayIdentifier — Display name of the variant control

character vector | string scalar

Optional display name of the variant control for measurement purpose in the calibration tool, which is different than the variant control name in the Simulink model, specified as a character vector or string scalar.

Data Types: char | string

Format — Display format of the variant control value

%[length].[layout]

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in CompuMethod of the variant control.

ParameterTuningService — Name of parameter tuning service interface

"Dictionary default" (default) | character vector | string scalar

Name of a parameter tuning service interface defined in the Embedded Coder Dictionary, specified as a character vector or string scalar. To use the dictionary default, specify "Dictionary default".

To configure the parameter tuning service interface, attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see "Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments".

Data Types: char | string

ParameterArgumentTuningService — Name of parameter argument tuning service interface

"Dictionary default" (default) | character vector | string scalar

Name of a parameter argument tuning service interface defined in the Embedded Coder Dictionary, specified as a character vector or string scalar. To use the dictionary default, specify "Dictionary default".

To configure the parameter argument tuning service interface, attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see "Configure

Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments”.

Data Types: `char` | `string`

Version History

Introduced in R2025a

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `getModelVariantControl` | `getModelVariantVariable` | `find` | `Simulink.VariantControl`

Topics

“Configure Code Generation for Variant Parameters in Model Workspace Using Code Mappings Editor”

“Generate Code for Instance-Specific Variation of Variant Parameter Values in Model Reference Hierarchy”

“Configure Parameters for C Code Generation”

“Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments”

setOutputport

Configure root-level outputport for code and calibration file (a2l) generation

Syntax

```
setOutputport(myCodeMappingObj,outputport,Name,Value)
```

Description

`setOutputport(myCodeMappingObj,outputport,Name,Value)` configures specified root-level Outputport blocks for code and calibration file (a2l) generation. Use this function to map specified root-level outputports to the storage class and storage class property settings that the code generator uses to produce C code. Or, set the calibration access for the outputport.

Examples

Configure Storage Class for Root-Level Outputports in Model to Model default

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for root Outputport blocks throughout the model to `Model default`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
outBlockHandles = find(cm,"Outputports")
setOutputport(cm,outBlockHandles,StorageClass="Model default");
```

Configure Storage Class and Storage Class Properties for Outputport

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for root-level outputport `Out1` to `ExportedGlobal`. Set the code identifier to `output1`, the definition file to `exOutSys.c`, and the header file to `exOutSys.h`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setOutputport(cm,"Out1",StorageClass="ExportedGlobal",Identifier="output1");
```

Configure Calibration Properties for Root-Level Outputports

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the calibration properties for root-level Outputport block `Out1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setOutputport(cm,"Out1",Export=true);
setOutputport(cm,"Out1",CalibrationAccess="Calibration");
setOutputport(cm,"Out1",CompuMethod="CM_Outputport");
```

```
setOutput(cm,"Out1",DisplayIdentifier="outport_display_name");
setOutput(cm,"Out1",Format="%4.2");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

outport — Name, path, or handle of root-level outport

character vector | string scalar | block handle | cell array of character vectors | cell array of string scalars | cell array of handles

Name, path, or handle of root-level outport to configure. To specify multiple outports, use a cell array.

Example: `"Out1"`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

`Auto` | `Bitfield` | `Dictionary default` | `ExportedGlobal` | `ExportToFile` | `GetSet` | `ImportedExtern` | `ImportedExternPointer` | `ImportFromFile` | `Localizable` | `Model default` | `Struct` | `Volatile` | storage class name

Storage class to set for the specified root Outport block. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Root-Level Outport Blocks for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the outport in the generated code.

Data Types: `char` | `string`

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`**HeaderFile — C header file**

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: `char` | `string`**Owner — Owner of global data**

character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`**PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays**`True` | `False`

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`**SetFunction — Name of set function**

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`**StructName — Name of structure**

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`**storageClassPropertyName — Value of storage class property**

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Export — Enable or disable generation of the output to calibration file (a2l)

true (default) | false

Enable this property to generate the output to calibration file (a2l).

BitMask — Extract single bits from the output value

hexadecimal value

Mask value in hexadecimal format to extract single bits from the output during calibration. This property is applicable only for integers.

Data Types: hexadecimal

CalibrationAccess — Enable or disable calibration of the output

NoCalibration (default) | Calibration

Select Calibration for the output to enable the calibration. Select NoCalibration to view the value of the output and disable the calibration.

CompuMethod — Name of conversion method

character vector | string scalar

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Data Types: char | string

DisplayIdentifier — Display name of the output

character vector | string scalar

Optional display name of the output for the measurement purpose in the calibration tool, which is different than the output name in the Simulink model.

Data Types: char | string

Format — Display format of the output value

%[length].[layout]

Special display format to be specified for measurement in the calibration tool. This format specification overrides the display format specified in CompuMethod of the output.

SenderService — Name of sender service interface

Dictionary default (default) | character vector | string scalar

String or character vector containing the name of a sender service interface defined in the Embedded Coder Dictionary. Within a target environment, a component sends data to other components by calling the target platform sender service. To use the dictionary default, specify "Dictionary default".

To configure the sender service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see "Configure Sender and Receiver Service Interfaces for Model Imports and Outputs".

Data Types: char | string

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getOutputport` | `setDataDefault`

Topics

“Configure Root-Level Output Blocks for C Code Generation”

“C Data Code Interface Configuration for Model Interface Elements”

“Configure Sender and Receiver Service Interfaces for Model Inports and Outports”

setSignal

Configure block signal data for code and calibration file (a2l) generation

Syntax

```
setSignal(myCodeMappingObj, portHandle, Name, Value)
```

Description

`setSignal(myCodeMappingObj, portHandle, Name, Value)` configures signals specified by block output ports for code and calibration file (a2l) generation. Use this function to map specified block output ports to the storage class and storage class property settings that the code generator uses to produce C code. Or, set the calibration access for the corresponding signal data.

This function does not apply to signals that originate from root-level Inport blocks. For signals that originate from root-level Inport blocks, see `setInport`.

Examples

Configure Storage Class for Block Output Signals

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for output signals of lookup table blocks `Table1` and `Table2D` to `ExportedGlobal`. After creating the object `cm` by calling function `coder.mapping.api.get`, get handles to the output ports for lookup table blocks. Set the storage class for the output signals by calling `setSignal`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
lut1_ports = get_param("ConfigurationRapidPrototypingInterface/Table1", "PortHandles");
lut2_ports = get_param("ConfigurationRapidPrototypingInterface/Table2", "PortHandles");
lut1_outPort = lut1_ports.Outport;
lut2_outPort = lut2_ports.Outport;
addSignal(cm, [lut1_outPort, lut2_outPort]);
setSignal(cm, [lut1_outPort, lut2_outPort], StorageClass="ExportedGlobal");
```

Configure Storage Class for Signal Data in Model Code Mappings to Model default

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for block output signals to `Model default`. After creating the object `cm` by calling function `coder.mapping.api.get`, get the port handles of the signal data in the code mappings. Set the storage class for the signals by calling `setSignal`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
portHandles = find(cm, "Signals");
setSignal(cm, portHandles, StorageClass="Model default");
```

Configure Code Identifiers for Block Output Signals

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the code identifiers for output signals of lookup table blocks `Table1` and `Table2` to `dout_Table1` and

dout_Table2. After creating the object `cm` by calling function `coder.mapping.api.get`, get handles to the output ports for lookup table blocks. Set the code identifiers for the output signals by calling `setSignal`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
lut1_ports = get_param("ConfigurationRapidPrototypingInterface/Table1", "PortHandles");
lut2_ports = get_param("ConfigurationRapidPrototypingInterface/Table2", "PortHandles");
lut1_outPort = lut1_ports.Outport;
lut2_outPort = lut2_ports.Outport;
addSignal(cm,[lut1_outPort,lut2_outPort]);
setSignal(cm,lut1_outPort,Identifier="dout_Table1D");
setSignal(cm,lut2_outPort,Identifier="dout_Table2D");
```

Configure Calibration Properties for Signal

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, configure the calibration properties for signal `Signal_1`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
lut1_ports = get_param("ConfigurationRapidPrototypingInterface/Table1", "PortHandles");
lut1_outPort = lut1_ports.Outport;
addSignal(cm,lut1_outPort);
Signal_1 = find(cm,"Signal");
setSignal(cm,Signal_1,Export=true);
setSignal(cm,Signal_1,CalibrationAccess="NoCalibration");
setSignal(cm,Signal_1,CompuMethod="CM_Signal_1");
setSignal(cm,Signal_1,DisplayIdentifier="Signal_display_name");
setSignal(cm,Signal_1,Format="%4.2");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: `myCM`

portHandle — Output port handle of signal source block

port handle | array of port handles

Signal to add to the code mappings, specified as a handle of an output port of the signal's source block. To specify multiple port handles, use an array.

Example: `portHandle`

Data Types: `port_handle` | array

Name-Value Pair Arguments

Specify 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 as `Name1`, `Value1`, ..., `NameN`, `ValueN`. The order of the name and value pair arguments does not matter.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

Auto | Bitfield | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified signals. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Signal Data for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the signal data in the generated code.

Data Types: char | string

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class GetSet.

Data Types: char | string

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes ExportToFile, GetSet, ImportFromFile, and Volatile.

Data Types: char | string

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes ExportToFile and Volatile.

Data Types: char | string

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

True | False

When model configuration parameter **Array layout** is set to Row-major, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a

multidimensional array. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`

SetFunction — Name of set function

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

storageClassName — Value of storage class property

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Data Types: `char` | `string`

Export — Enable or disable generation of the signal to calibration file (a2I)

`true` (default) | `false`

Enable this property to generate the signal to calibration file (a2I).

BitMask — Extract single bits from the signal value

hexadecimal value

Mask value in hexadecimal format to extract single bits from the signal during calibration. This property is applicable only for integers.

Data Types: `hexadecimal`

CalibrationAccess — Enable or disable calibration of the signal

`NoCalibration` (default) | `Calibration`

Select `Calibration` for the signal to enable the calibration. Select `NoCalibration` to view the value of the signal and disable the calibration.

CompuMethod — Name of conversion method

character vector | string scalar

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Data Types: `char` | `string`

DisplayIdentifier — Display name of the signal

character vector | string scalar

Optional display name of the signal for the measurement purpose in the calibration tool, which is different than the signal name in the Simulink model.

Data Types: `char` | `string`

Format — Display format of the signal value

`%[length].[layout]`

Special display format to be specified for measurement in the calibration tool. This format specification overrides the display format specified in `CompuMethod` of the signal.

MeasurementService — Name of measurement service interface

`Dictionary default` (default) | `character vector` | `string scalar`

String or character vector containing the name of a measurement service interface defined in the Embedded Coder Dictionary. By configuring the measurement service interface for signals, states, and data stores, you can preserve the data in the generated code for measurement. To use the dictionary default, specify `"Dictionary default"`. If data from the state does not need to be preserved in the code, specify `"Not measured"`.

To configure the measurement service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see “Configure Measurement Service Interfaces for Signals, States, and Data Stores”.

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`addSignal` | `coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getSignal` | `removeSignal` | `setDataDefault`

Topics

“Configure Signal Data for C Code Generation”

“C Data Code Interface Configuration for Model Interface Elements”

“Configure Measurement Service Interfaces for Signals, States, and Data Stores”

setState

Configure block states for code and calibration file (a2l) generation

Syntax

`setState(myCodeMappingObj,block,Name,Value)`

Description

`setState(myCodeMappingObj,block,Name,Value)` configures specified block states for code and calibration file (a2l) generation. Use this function to map specified block states to the storage class and storage class property settings that the code generator uses to produce C code. Or, set the calibration access for the for the states.

Examples

Configure Storage Class for Block State

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the storage class for the state X of Unit Delay block Delay to `ExportedGlobal`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
setState(cm,"ConfigurationRapidPrototypingInterface/Delay",StorageClass="ExportedGlobal");
```

Configure Storage Class for Block States in Model to Model default

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, configure the storage class for block states throughout the model to `Model default`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
blockHandles = find(cm,"States");
setState(cm,blockHandles,StorageClass="Model default");
```

Configure Code Identifier for Block State

In the model code mappings for model `ConfigurationRapidPrototypingInterface`, configure the code identifier for the state X of Unit Delay block Delay to `dstate_X`.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
blockHandles = find(cm,"States");
setState(cm,blockHandles,StorageClass="Model default");
setState(cm,blockHandles,Identifier="dstate_X");
```

Configure Calibration Properties for State of Unit Delay Block

From the model code mappings for model `ConfigurationRapidPrototypingInterface`, set the calibration properties configured for state X of Unit Delay block Delay.

```
openExample("ConfigurationRapidPrototypingInterface");
cm = coder.mapping.api.get("ConfigurationRapidPrototypingInterface");
state = find(cm,"State");
setState(cm,state,Export=true);
setState(cm,state,CalibrationAccess="Calibration");
setState(cm,state,CompuMethod="CM_state");
setState(cm,state,DisplayIdentifier="state_X");
setState(cm,state,Format="%4.2");
```

Input Arguments

myCodeMappingObj — Code mapping object

CodeMapping object

Code mapping object (model code mappings) returned by a call to function `coder.mapping.api.get`.

Example: myCM

block — Path or handle of block

character vector | string scalar | block handle | cell array of character vectors | cell array of string scalars | cell array of block handles

Path or handle of the block containing the state to configure. To specify multiple block states, use a cell array.

Example: blockHandle

Data Types: char | string | block_handle | cell

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `StorageClass="ExportedGlobal"`

StorageClass — Name of storage class

Auto | Bitfield | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified block state. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. See “Configure Block States for C Code Generation”.

Identifier — Name of variable

character vector | string scalar

Name for the variable that represents the block state in the generated code.

Data Types: `char` | `string`

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

GetFunction — Name of get function

character vector | string scalar

Name of a `get` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, and `Volatile`.

Data Types: `char` | `string`

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `ExportToFile` and `Volatile`.

Data Types: `char` | `string`

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

`True` | `False`

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes `ExportToFile`, `GetSet`, `ImportFromFile`, `Localizable`, and `Volatile`.

Data Types: `logical`

SetFunction — Name of set function

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Data Types: `char` | `string`

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Data Types: `char` | `string`**storageClassPropertyName — Value of storage class property**

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Export — Enable or disable generation of the block state to calibration file (a2l)`true` (default) | `false`

Enable this property to generate the block state to calibration file (a2l).

BitMask — Extract single bits from the block state value

hexadecimal value

Mask value in hexadecimal format to extract single bits from the block state during calibration. This property is applicable only for integers.

Data Types: `hexadecimal`**CalibrationAccess — Enable or disable calibration of the block state**`NoCalibration` (default) | `Calibration`

Select `Calibration` for the block state to enable the calibration. Select `NoCalibration` to view the value of the block state and disable the calibration.

CompuMethod — Name of conversion method

character vector | string scalar

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Data Types: `char` | `string`**DisplayIdentifier — Display name of the block state**

character vector | string scalar

Optional display name of the block state for the measurement purpose in the calibration tool, which is different than the block state name in the Simulink model.

Data Types: `char` | `string`**Format — Display format of the block state value**`%[length].[layout]`

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in `CompuMethod` of the state.

MeasurementService — Name of measurement service interface`Dictionary default` (default) | character vector | string scalar

String or character vector containing the name of a measurement service interface defined in the Embedded Coder Dictionary. By configuring the measurement service interface for signals, states, and data stores, you can preserve the data in the generated code for measurement. To use the dictionary default, specify "Dictionary default". If data from the state does not need to be preserved in the code, specify "Not measured".

To configure the measurement service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see "Configure Measurement Service Interfaces for Signals, States, and Data Stores".

Data Types: `char` | `string`

Version History

Introduced in R2020b

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.api.get` | `find` | `getDataDefault` | `getState` | `setDataDefault`

Topics

"Configure Block States for C Code Generation"

"C Data Code Interface Configuration for Model Interface Elements"

"Configure Measurement Service Interfaces for Signals, States, and Data Stores"

coder.mapping.utils.create

Create code mappings object for configuring data and function interface for C and C++ code generation

Syntax

```
coderMapObj = coder.mapping.utils.create(simulinkModel)
coderMapObj = coder.mapping.utils.create(simulinkModel,configObj)
```

Description

`coderMapObj = coder.mapping.utils.create(simulinkModel)` creates a code mappings environment for the specified model, if one does not exist, and returns it as the object `coderMapObj`. Code mappings associate model data elements and functions with configurations for C or C++ code generation. If code mappings exist for the specified model, the function returns those code mappings as the object `coderMapObj`.

`coderMapObj = coder.mapping.utils.create(simulinkModel,configObj)` imports default memory section and shared utility naming rule configurations from configuration set `configObj` while creating C code mappings for the specified model. See “Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings”.

Examples

Programmatically Create and Use Code Mappings of Simulink Models

Use the programmatic interface to create and use code mappings objects of Simulink models.

Load the model `NoCoderMapping`.

```
simulinkModel = "NoCoderMapping";
load_system(simulinkModel);
```

Use a try-catch block to determine whether a code mappings object exists for the model. Inside the try block, attempt to retrieve the existing object by using function `coder.mapping.api.get`. Inside the catch block, create a new code mappings object by using the function `coder.mapping.utils.create`. Store the code mappings object in the variable `codeMapObj`.

Add print messages to show whether the model had an existing code mappings object or not.

```
try
    codeMapObj = coder.mapping.api.get(simulinkModel);
    fprintf(" " + ...
        " =====\n" + ...
        " The model already had code mappings.\n" + ...
        " =====\n");
catch
    fprintf(" " + ...
        " =====\n" + ...
        " The model does not have code mappings.\n" + ...
```

```

        " Creating new code mappings for the model.\n" + ...
        " =====\n");
codeMapObj = coder.mapping.utils.create(simulinkModel);
end

```

```

=====
The model does not have code mappings.
Creating new code mappings for the model.
=====

```

Retrieve the storage class of input port `inport_1` of the model.

```
getInport(codeMapObj,"inport_1","StorageClass")
```

```
ans =
'Auto'
```

Set the storage class of input port `inport_1` to `ExportedGlobal`.

```
setInport(codeMapObj,"inport_1",StorageClass="ExportedGlobal")
```

Use the same try-catch block to see how the function `coder.mapping.api.get` is now able to retrieve the existing code mappings of the model.

```

try
    codeMapObj = coder.mapping.api.get(simulinkModel);
    fprintf(" " + ...
        " =====\n" + ...
        " The model already had code mappings.\n" + ...
        " =====\n");
catch
    fprintf(" " + ...
        " =====\n" + ...
        " The model does not have code mappings.\n" + ...
        " Creating new code mappings for the model.\n" + ...
        " =====\n");
    codeMapObj = coder.mapping.utils.create(simulinkModel);
end

```

```

=====
The model already had code mappings.
=====

```

Retrieve the storage class of input port `inport_1` of the model. Notice that it is the storage class that you set previously.

```
getInport(codeMapObj,"inport_1","StorageClass")
```

```
ans =
'ExportedGlobal'
```

Close the model without saving it.

```
close_system(simulinkModel,false)
```

Input Arguments

simulinkModel — Name of model

handle | character vector | string scalar

Simulink model for which to create and return a code mappings object, specified as a handle or a character vector or string scalar with the model name (without the `.slx` file extension), or the relative or absolute path of the model file (including the `.slx` file extension).

Note The model must be loaded (for example, by using `load_system`).

Example: `"configModel"`

Data Types: `char` | `string` | `handle`

configObj — Configuration object

handle | character vector | string scalar

Model configuration object from which to import memory section and shared utility naming rule configurations while creating code mappings for `simulinkModel`, specified as a handle, character vector or string scalar. 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.

Example: `"my_legacyConfigset"`

Output Arguments

coderMapObj — Code mapping object

`CodeMapping` object | `CodeMappingCPP` object

The model code mappings, returned as a `CodeMapping` object or a `CodeMappingCPP` object.

Output	Input Object
<code>coder.mapping.api.CodeMapping</code>	Simulink model configured for C code generation
<code>coder.mapping.api.CodeMappingCPP</code>	Simulink model configured for C++ code generation

Version History

Introduced in R2020b

See Also

`coder.mapping.api.get` | `coder.mapping.api.CodeMapping` | `coder.mapping.api.CodeMappingCPP`

Topics

"C Data Code Interface Configuration for Model Interface Elements"
 "Programmatically Configure C++ Interface"

`coder.mapping.api.CoderDictionary`

Query and set the code settings of dictionary defaults in an Embedded Coder dictionary within a Simulink data dictionary

Description

A coder dictionary code mappings object and its related functions enable you to configure C code generation settings for dictionary defaults in an Embedded Coder dictionary within a Simulink data dictionary. For model data categories, code mappings associate data categories with configurations that consist of a storage class and storage class properties. For functions, code mappings associate function categories with configurations that consist of a function customization template. Reduce the effort of preparing a model for code generation by specifying default configurations for categories of data elements and functions across a model.

Creation

Syntax

```
myCoderDictionaryObj = coder.mapping.api.get(dictionary)
myCoderDictionaryObj = coder.mapping.utils.create(dictionary)
```

Description

`myCoderDictionaryObj = coder.mapping.api.get(dictionary)` returns the active code mappings for the specified dictionary as object `myCoderDictionaryObj`. Code mappings associate data elements and functions in the data dictionary with configurations for code generation.

If code mappings do not exist, Simulink returns an error. You can create a code mappings object with a call to `coder.mapping.utils.create`.

`myCoderDictionaryObj = coder.mapping.utils.create(dictionary)` creates a code mappings environment for the specified data dictionary and returns the mappings as object `myCoderDictionaryObj`. Code mappings associate data elements and functions with configurations for C or C++ code generation. If code mappings exist for the specified data dictionary, the function returns those code mappings.

Input Arguments

dictionary — Name of data dictionary

character vector | string scalar

Data dictionary for which to return code mappings object, specified as a character vector or string scalar representing the dictionary name.

Example: "exCodeDefs.sldd"

Data Types: char | string

Object Functions

<code>getDataDefault</code>	Get default code settings for data category
<code>setDataDefault</code>	Set default code settings for data category
<code>getFunctionDefault</code>	Get default function customization template or memory section for model functions category
<code>setFunctionDefault</code>	Set default function customization template and memory section for model functions category

Examples

Create Environment to Configure Code Mappings for Data Dictionary

For the data dictionary `exCodeDefs.sldd`, create the environment for configuring the data and functions for code generation. After calling this function, use calls to other functions listed under Object Functions to configure aspects of code generation for the interface elements.

```
coder.mapping.utils.create("exCodeDefs.sldd");
```

Version History

Introduced in R2021a

See Also

`coder.mapping.api.CodeMapping` | `coder.mapping.utils.create` |
`coder.mapping.api.CodeMappingCPP`

Topics

"C Data Code Interface Configuration for Model Interface Elements"

"Programmatically Configure C++ Interface"

setDataDefault

Set default code settings for data category

Syntax

```
setDataDefault(myCoderDictionaryObj,category,Name,Value)
```

Description

`setDataDefault(myCoderDictionaryObj,category,Name,Value)` sets the default storage class and storage class property values in the code mappings for the specified category of model data.

Examples

Configure default code settings for a data category in a data dictionary

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get('codeDefinitions.sldd');
```

To see the storage class of root-level inports for the dictionary, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =
```

```
    'Default'
```

The dictionary uses the default storage class for inports.

To configure the storage class, use the `setDataDefault` function.

```
setDataDefault(cm, 'Inports', 'StorageClass', 'ExportedGlobal')
```

To verify that the storage class of inports is now set to `ExportedGlobal`, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =
```

```
    'ExportedGlobal'
```

Input Arguments

myCoderDictionaryObj — Coder dictionary object

`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

category — Model data element category

Constants | ExternalParameterObjects | GlobalDataStores | Inports | InternalData | ModelParameters | ModelParameterArguments | Outports | SharedLocalDataStores

Category of data elements to return a property value for.

Example: 'Inports'

Name-Value Pair Arguments

Specify 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 as Name1, Value1, ..., NameN, ValueN. The order of the name and value pair arguments does not matter.

Example: 'StorageClass', 'ExportedGlobal'

StorageClass — Name of storage class

Auto | Bitfield | CompileFlag | Const | ConstVolatile | Define | Dictionary default | ExportedGlobal | ExportToFile | FileScope | GetSet | ImportedDefine | ImportedExtern | ImportedExternPointer | ImportFromFile | Localizable | Model default | Struct | Volatile | storage class name

Storage class to set for the specified data element category. The name of a predefined storage class or storage class that is defined in the Embedded Coder Dictionary associated with the model. Values that you can specify vary depending on the category that you specify. See “Choose Storage Class for Controlling Data Representation in Generated Code”.

Example: 'StorageClass', 'ImportedExtern'

DefinitionFile — C source file

character vector | string scalar

File name for a C source file that contains definitions for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, ExportToFile, and Volatile.

Example: 'DefinitionFile', 'myDataDefs.c'

Data Types: char | string

GetFunction — Name of get function

character vector | string scalar

Name of a get function that a data element calls in the generated code. Applies to storage class GetSet.

Example: 'GefFunction', 'myDataGetFunction'

Data Types: char | string

HeaderFile — C header file

character vector | string scalar

File name for a C header file that contains declarations for global data read by data elements and external code. Applies to storage classes Const, ConstVolatile, Define, ExportToFile, GetSet, ImportedDefine, ImportFromFile, and Volatile.

Example: 'HeaderFile', 'myDataDecl.h'

Data Types: `char` | `string`

MemorySection — Name of memory section

character vector | string scalar

Name of a memory section that is defined in the Embedded Coder Dictionary associated with the model.

Example: `'MemorySection','myFastMem'`

Data Types: `char` | `string`

Owner — Owner of global data

character vector | string scalar

Name of the model that owns global data, which is used by other models in the same model hierarchy. The code generated for the model that owns the data includes the global data definition. Applies to storage classes `Const`, `ConstVolatile`, `ExportToFile`, and `Volatile`.

Example: `'Owner','myModelA'`

Data Types: `char` | `string`

PreserveDimensions — Boolean flag indicating whether to preserve dimensions of multidimensional arrays

`True` | `False`

When model configuration parameter **Array layout** is set to `Row-major`, a flag that indicates whether to preserve dimensions of a data element represented in generated code as a multidimensional array. Applies to storage classes `Const`, `ConstVolatile`, `ExportToFile`, `FileScope`, `ImportFromFile`, `Localizable`, and `Volatile`.

Example: `'PreserveDimensions','True'`

Data Types: `logical`

SetFunction — Name of set function

character string | string scalar

Name of a `set` function that a data element calls in the generated code. Applies to storage class `GetSet`.

Example: `'SetFunction','myDataSetFunction'`

Data Types: `char` | `string`

StructName — Name of structure

character vector | string scalar

Name that the code generator uses to identify the structure for a data element in the generated code. Applies to storage classes `Bitfield` and `Struct`.

Example: `'StructName','myDataStruct'`

storageClassPropertyName — Value of storage class property

depends on property definition

Storage class property defined in the Embedded Coder Dictionary. Values that you can specify vary depending on the storage class definition.

Data Types: `char` | `string`

Version History

Introduced in R2021a

See Also

`coder.mapping.api.CoderDictionary` | `getDataDefault` | `setFunctionDefault` | `getFunctionDefault`

getDataDefault

Get default code settings for data category

Syntax

```
value = getDataDefault(myCoderDictionaryObj, category, property)
```

Description

`value = getDataDefault(myCoderDictionaryObj, category, property)` returns the value from the code mappings of the specified property for the specified data category.

Examples

Configure default code settings for a data category in a data dictionary

Use the `coder.mapping.api.get` function to access the `CoderDictionary` object associated with the data dictionary.

```
cm = coder.mapping.api.get('codeDefinitions.slidd');
```

To see the storage class of root-level inports for the dictionary, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =  
    'Default'
```

The dictionary uses the default storage class for inports.

To configure the storage class, use the `setDataDefault` function.

```
setDataDefault(cm, 'Inports', 'StorageClass', 'ExportedGlobal')
```

To verify that the storage class of inports is now set to `ExportedGlobal`, use the `getDataDefault` function.

```
value = getDataDefault(cm, 'Inports', 'StorageClass')
```

```
value =  
    'ExportedGlobal'
```

Input Arguments

myCoderDictionaryObj — Coder dictionary object

`CoderDictionary` object

Coder dictionary object returned by a call to function `coder.mapping.api.get`.

category — Model data element category

Constants | ExternalParameterObjects | GlobalDataStores | Inports | InternalData | ModelParameters | ModelParameterArguments | Outports | SharedLocalDataStores

Category of data elements to return a property value for.

Example: 'Inports'

property — Code mapping property value to return

StorageClass | Identifier | DefinitionFile | GetFunction | HeaderFile | MemorySection | Owner | PreserveDimensions | SetFunction | StructName | storage class property name

Code mapping property that you return a value for. Specify one of these property names or a property name for a storage class defined in the Embedded Coder Dictionary .

Information to Return	Property Name
Name of storage class	StorageClass
Name of variable for data element in the generated code	Identifier
Name of source definition file that contains definitions for global data that is read by the data element and external code	DefinitionFile
Name of <code>get</code> function called by code generated for the data element	GetFunction
Name of source header file that contains declarations for global data that is read by the model data element and external code	HeaderFile
Character vector or string scalar that names a memory section for a model defined in the Embedded Coder Dictionary.	MemorySection
Name of model for which the code generator places the definition for data element shared by multiple models in a model hierarchy	Owner
Boolean value indicating whether code generator preserves dimensions of data that is represented as a multidimensional array	PreserveDimensions
Name of <code>set</code> function called by code generated for data element	SetFunction
Name of structure in generated code for data element	StructName

Example: 'Identifier'

Output Arguments**value — Code mapping property value of category**

character vector

The code mapping property value of the specified category, returned as a character vector.

Version History

Introduced in R2021a

See Also

`coder.mapping.api.CoderDictionary` | `setDataDefault` | `getFunctionDefault` | `setFunctionDefault`

`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 `CounterModel`, close the report.

```
coder.report.close()
```

Version History

Introduced in R2012a

See Also

`coder.report.open` | `coder.report.generate`

Topics

“Reports for Code Generation”

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 `CounterModel`.

```
openExample('CounterModel.slx')
```

Build the model. The model is configured to create and open a code generation report.

```
slbuild('CounterModel');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report.

```
coder.report.generate('CounterModel');
```

Generate Code Generation Report for Subsystem

Open the model `CounterModel`.

```
openExample('CounterModel.slx')
```

Build the subsystem. The model is configured to create and open a code generation report.

```
slbuild('CounterModel/Amplifier');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report for the subsystem.

```
coder.report.generate('CounterModel/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 RollAxisAutopilot.

```
model = 'RollAxisAutopilot';
openExample('ecoder/TraceGeneratedCodeToBlocksExample','supportingFile',model)
```

Build the model. The model is configured to create and open a code generation report.

```
slbuild(model);
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report that includes the static code metrics report.

```
coder.report.generate(model,...
'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: 'CounterModel'

Data Types: char

subsystem — Subsystem name

character vector

Subsystem name specified as a character vector

Example: 'CounterModel/Amplifier'

Data Types: char

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

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.

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.

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.

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.

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.

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.

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.

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 Generate static code metrics.

Example: `'GenerateCodeMetricsReport','on'`

Data Types: char

Version History

Introduced in R2012a

See Also

`coder.report.open` | `coder.report.close`

Topics

"Reports for Code Generation"

"Configure and Generate Code Generation Report Programmatically"

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 `CounterModel`, open a code generation report for the model.

```
coder.report.open('CounterModel')
```

Open code generation report for a subsystem

Open a code generation report for the subsystem 'Amplifier' in model 'CounterModel'.

```
coder.report.open('CounterModel/Amplifier')
```

Input Arguments

model — Model name

character vector

Model name specified as a character vector

Example: 'CounterModel'

Data Types: char

subsystem — Subsystem name

character vector

Subsystem name specified as a character vector

Example: 'CounterModel/Amplifier'

Data Types: char

Version History

Introduced in R2012a

See Also

`coder.report.close` | `coder.report.generate`

Topics

"Reports for Code Generation"

"Generate Code Generation Report"

coder.cdf.export

Generate CDF (Calibration Data Format) file according to ASAM AE CDF standards

Syntax

```
coder.cdf.export(modelName)
coder.cdf.export(modelName,Name=Value)
```

Description

`coder.cdf.export(modelName)` generates a CDF file for `modelName`.

`coder.cdf.export(modelName,Name=Value)` specifies additional options for CDF file creation with one or more Name, Value pair arguments. For example, you can specify a location where to save the CDF file.

Examples

Generate CDF File for Model

Generate a CDF file for the selected model and save it in the build folder of the model.

```
% Generate CDF file for model
coder.cdf.export('modelName')
```

Generate CDF File and Save it with a Custom Name

Generate a CDF file for the selected model and save it with the custom name specified.

```
% Export CDF file and save it as
coder.cdf.export('modelName',FileName='test_car')
```

Generate CDF File at Specified Location

Generate a CDF file for the selected model and save it in the specified folder.

```
% Export CDF file to specified path
coder.cdf.export('modelName',Folder='/home/temp/workspace/')
```

Generate CDF File with Specified Schema Type

Generate a CDF file of the specified schema type for the selected model.

```
% Export CDF file with dtd schema type
coder.cdf.export('modelName', SchemaType='DTD')
```

Input Arguments

modelName — name of the model

character vector | string scalar

Name of the model.

Example: 'MyModel', 'nav_app'

Name-Value Pair Arguments

Specify optional pairs of arguments as **Name1=Value1, ..., NameN=ValueN**, where **Name** is the argument name and **Value** is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: Folder='/home/applications' generate a CDF file for the model and saves it in a specified folder.

Folder — Folder location for exported CDF file

character vector | string scalar

Full path to a folder in which to place an exported CDF file.

Example: Folder='/home/temp/prjct/'

FileName — Custom name for the exported CDF file

character vector | string scalar

Name for the CDF file to save it as in the folder.

Example: FileName='test_car'

SchemaType — Schema type for CDF file

DTD (default) | XSD

Schema type for the CDF file can be XSD (XML Schema Definition) or DTD (Document Type Definition).

Example: SchemaType='DTD'

Version History

Introduced in R2021a

See Also

`coder.asap2.export` | Generate Calibration Files

Topics

“Generate ASAP2 and CDF Calibration Files”

coder.asap2.export

Generate ASAP2 (A2L) file according to ASAM MCD-2 MC standards

Syntax

```
coder.asap2.export(modelName)
coder.asap2.export(modelName,Name=Value)
```

Description

`coder.asap2.export(modelName)` generates an ASAP2 (A2L) file for `modelName`. The ASAP2 (A2L) file generated for a top model contains information from the referenced models.

`coder.asap2.export(modelName,Name=Value)` specifies additional options for ASAP2 (A2L) creation with one or more "Name=Value" arguments. For example, you can specify a location where to save the ASAP2 file. You can provide the symbol file of the model to replace ECU addresses in the ASAP2 file.

Examples

Generate ASAP2 File for Model

Generate an ASAP2 file for the selected model and save it in the build folder of the model by using this command.

```
% Generate ASAP2 file for model
coder.asap2.export('modelName')
```

Generate ASAP2 File by Using Customizations

Generate an ASAP2 file by modifying the name of ASAP2 file, location where to save it, version of ASAP2 to be generated, and excluding comments by using this command.

```
% Export ASAP2 file and save it as
coder.asap2.export('modelName',FileName='test_car')

% Export ASAP2 file to specified path
coder.asap2.export('modelName',Folder='/home/temp/workspace/')

% Generate ASAP2 file with version 1.71
coder.asap2.export('modelName',Version='1.71')

% Generate ASAP2 file with comments excluded
coder.asap2.export('modelName',Comments=false)
```

Generate ASAP2 File for Model by Using Symbol File

Generate an ASAP2 file for the selected model that has ECU addresses based on the ELF symbol file associated with the executable by using this command.

```
% Generate ASAP2 file for model
coder.asap2.export('modelName',MapFile='model.elf')
```

Exclude A2ML and IF_DATA Sections from Generated ASAP2 File

Generate an ASAP2 file for the selected model and exclude the A2ML and IF_DATA sections by using this command.

```
% Generate ASAP2 file with A2ML and IF_DATA excluded
coder.asap2.export('modelName',GenerateXCPInfo=false)
```

Generate ASAP2 File by Using Custom Model Class Instance Name

Specify the name of the model class instance. The objName is declared in the global namespace by using this command.

```
% Use custom specified name as object name in ASAP2 file
coder.asap2.export('modelName',ModelClassName=objName)

% Specify the name of model class instance declared inside the namespace. Here instance customObj
% is declared in customNameSpace
coder.asap2.export('modelName',ModelClassName='customNamespace::customObj')
```

Generate ASAP2 File by Using Customized ASAP2 Fields

Create a custom base object and specify the fields. Customize the contents of the ASAP2 file by using a custom base object by using this command.

```
% Create custom base object and provide fields you want to modify
obj = coder.asap2.UserCustomizeBase;
obj.HeaderComment = 'Header comment';
obj.ModParComment = 'Mod Par comment';
obj.ModCommonComment = 'Mod Common comment';
obj.ASAP2FileName = 'File name';
obj.ByteOrder = 'BYTE_ORDER_MSB_LAST';

% Generate ASAP2 file with custom base created
coder.asap2.export('modelName',CustomizationObject=obj);
```

Generate ASAP2 File by Grouping Parameters and Measurements

Group the parameters and measurements into arrays, scalars, maps, and curves by using this command.

```
coder.asap2.export('modelName',CustomizeGroupsBy={'ARRAY','SCALAR','MAP','CURVE'});
```


Export Record Layouts

Export the record layouts of the base data types to a file named `RecordLayouts.a2l` by using this command.

```
coder.asap2.export('modelName', IncludeAllRecordLayouts=true)
```

Generate ASAP2 File by Excluding Structures

Exclude structure elements by using this command.

```
coder.asap2.export('modelName', SupportStructureElements=false);
```

Generate ASAP2 File by Excluding Referenced Model Elements

Exclude referenced model elements by using this command.

```
coder.asap2.export('modelName', IncludeReferencedModels=false);
```

Generate ASAP2 File by Excluding 64-Bit Elements

Exclude 64-bit integers by using this command.

```
coder.asap2.export('modelName', Support64bitIntegers=false);
```

Generate ASAP2 File by Specifying ECU Address Extension

Specify additional address information in the ASAP2 file by using this command.

```
coder.asap2.export('modelName', EcuAddressExtension=4);
```

Generate ASAP2 File Using the Saved Preferences

Generate the ASAP2 file using the settings updated by using this command.

```
coder.asap2.export('modelName', UseSavedSettings=false);
```

Generate ASAP2 File by Excluding RTE Elements

Exclude AUTOSAR RTE elements in the ASAP2 file for a AUTOSAR classic model by using this command.

```
coder.asap2.export('autosarclassic-modelName', IncludeAutosarRteElements=false);
```

Generate ASAP2 File by Updating Array Layout

Export ASAP2 file by modifying the array layout to view the array data in required format by using this command.

```
coder.asap2.export('modelName', ToggleArrayLayout=true);
```

Generate Separate ASAP2 Files for Record Layouts and CompuMethods

Export record layouts and CompuMethods as separate ASAP2 files for an embedded coder model by using this command.

```
coder.asap2.export('ert-modelName', IncludeSharedElements=true);
```

Generate ASAP2 File by Excluding Variant Parameters

Exclude variant parameters in ASAP2 file by using this command.

```
coder.asap2.export('modelName', IncludeVariantCoding=false);
```

Input Arguments

modelName — name of the model

character vector | string scalar

Name of the model.

Example: 'MyModel', 'nav_app'

Name-Value Pair Arguments

Specify optional pairs of arguments as **Name1=Value1, ..., NameN=ValueN**, where **Name** is the argument name and **Value** is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `MapFile='model.elf'` generates an ASAP2 file for the model containing debug mapping information from the `model.elf` file.

Comments — Include comments in ASAP2 file

true (default) | false

Generate the ASAP2 file by including or excluding comments.

Example: `Comments=true`

CustomEcuDescriptions — Update the ECU descriptions

ECU description object

Update the current ECU description in the ASAP2 file to custom values by creating and providing a description object.

Note This argument cannot be used with other Name=Value pairs.

Example: CustomEcuDescriptions=description_object

CustomizationObject — Customize ASAP2 fields

coder.asap2.UserCustomizeBase object (default)

Create a user base and customize the ASAP2 fields such as:

- ASAP2FileName
- ByteOrder
- HeaderComment
- ModParComment
- ModCommonComment

Example: CustomizationObject=obj

CustomizeGroupsBy — Customize groups in ASAP2 file

ARRAY | ASCII | SCALAR | MAP | CUBOID | CURVE | CUBE_4 | CUBE_5

Group the parameters and signals based on their properties.

Example: CustomizeGroupsBy={'ARRAY', 'MAP'}

EcuAddressExtension — Specify additional address information

int

Specify a 16-bit integer value to specify additional address information in the ASAP2 file. By default, no additional address information is required.

Example: EcuAddressExtension=4

FileName — Custom name for the exported ASAP2 file

character vector | string scalar

Name for the exported ASAP2 file.

Example: FileName='test_car'

Folder — Folder location to export ASAP2 file

character vector | string scalar

Specify export location for the generated ASAP2 file.

Example: Folder='/home/temp/prjct/'

GenerateXCPIInfo — Include A2ML and IF_DATA in ASAP2 file

true (default) | false

Generate the ASAP2 file by including or excluding A2ML and IF_DATA sections.

Example: GenerateXCPIInfo=true

IncludeAllRecordLayouts — Export record layouts of the base data types

false (default) | true

Export record layouts of all the base data types to a file named `RecordLayouts.a2l` according to the version of the ASAP2 file.

Example: `IncludeAllRecordLayouts=true`

IncludeAutosarRteElements — Export RTE elements

true (default) | false

Generate the ASAP2 file by including or excluding the AUTOSAR RTE elements.

Note This option is applicable only for AUTOSAR classic models.

Example: `IncludeAutosarRteElements=true`

IncludeDefaultEventList — Create default event

true (default) | false

Specify true to include a default event list in the ASAP2 file.

Example: `IncludeDefaultEventList=true`

IncludeReferencedModels — Export referenced model elements

true (default) | false

Specify true to include referenced model elements in ASAP2 file. Specify false to exclude.

Example: `IncludeReferencedModels=true`

IncludeSharedElements — Export CompuMethods and record layouts separately

true (default) | false

Generate separate ASAP2 files for CompuMethods and record layouts.

Note This option is applicable for models with ERT - based system target files.

Example: `IncludeSharedElements=true`

IncludeVariantCoding — Export variant parameters

true (default) | false

Specify true to export variant parameters to ASAP2 file. Specify false to skip the variant parameters.

Example: `IncludeVariantCoding=true`

IndentFile — Follow indentation in ASAP2 file

false (default) | true

Generate an ASAP2 file by following indentation.

Example: `IndentFile=true`

MapFile — Name of symbol file for model

ELF | PDB | DWARF

Name of the model symbol file that contains symbols of generated code. For example, the addresses of variables used in generated code.

Example: `MapFile='model.elf'`

ModelClassName — Specify class instance and path names

character vector | string scalar

Custom model instance name in an ASAP2 file. This argument is applicable only for AUTOSAR adaptive models.

Example: `ModelClassName='customObj'` or
`ModelClassName='customNameSpace::customObj'`

Support64bitIntegers — Include or exclude the 64-bit integers in ASAP2 file

true (default) | false

Specify false to exclude 64-bit integers in the generated ASAP2 file.

Example: `Support64bitIntegers=false`

SupportStructureElements — Include or exclude the structure elements in ASAP2 file

true (default) | false

Specify false to exclude elements that are part of the structure in the generated ASAP2 file.

Example: `SupportStructureElements=false`

ToggleArrayLayout — Modify the array layout

false (default) | true

Update the array layout to ROW_DIR or COL_DIR to view the array data in row wise or column wise.

Example: `ToggleArrayLayout=true`

UseSavedSettings — Save and use the preferences

false (default) | true

Save the preferences made and generate the ASAP2 file with the preferences updated such as including the comments, version of the ASAP2 file.

Example: `UseSavedSettings=false`

Version — Version of ASAP2 file

1.71 (default) | 1.31 | 1.61

ASAP2 file format based on the ASAM MCD-2 MC standard defined by ASAM. There are multiple versions of the ASAM MCD-2 MC standard. Specify the version of ASAP2 that you want.

Example: `Version='1.61'` or `Version='1.31'`

Version History

Introduced in R2021a

R2024a: Group Parameters and Signals of Type ASCII

The function now enables you to export the parameters and signals of ASCII data type as a group.

R2023a: Applicable for DDS Blockset Models

The function is now applicable for DDS Blockset models.

R2022b: Applicable for SLRT Models

The function is now applicable for Simulink Real-Time models. Additional name=value pairs are added to save the preferences of ASAP2 file generation, include the default event list, and toggle array layout.

For AUTOSAR classic models, use the function to include/exclude RTE elements.

R2022a: Additional Name=Value pairs

Additional name=value pairs are added to include/exclude structure elements, 64-bit integers, and add custom ECU descriptions to the ASAP2 file.

See Also

`coder.cdf.export` | Generate Calibration Files | `coder.asap2.getEcuDescriptions`

Topics

“Generate ASAP2 and CDF Calibration Files”

“Customize Generated ASAP2 File”

“Customize Using Base Files”

“Create and Add Custom Data Elements”

“Frequently Asked Questions About ASAP2 File Generation”

coder.asap2.merge

Merge data elements from multiple ASAP2 (A2L) files

Syntax

```
mergeObj = coder.asap2.merge(modelName,a2lFilePath)
```

Description

`mergeObj = coder.asap2.merge(modelName,a2lFilePath)` merges the data elements from the source A2L file specified in the `a2lFilePath` with the ECU descriptor object of Simulink model `modelName`. You can merge more than one A2L files with the Simulink model.

Note To use the `coder.asap2.merge` function, Vehicle Network Toolbox™ is required.

Examples

Merge ASAP2 Files

Open a model and build it. The model can be dummy without any data elements.

```
open_system("ModelName1");
slbuild(bdroot);
```

Call the function by specifying the A2L files to be merged.

```
mergeObj = coder.asap2.merge("ModelName1",{"ASAP2File1.a2l","ASAP2File2.a2l"});
```

The `mergeObj` descriptor object for the model `ModelName1` contains the data elements from the ASAP2 files `ASAP2File1` and `ASAP2File2`.

Export the ASAP2 file for the model by using the command.

```
coder.asap2.export("ModelName1",CustomEcuDescriptions=mergeObj)
```

Input Arguments

modelName — Simulink model name

char vector | string scalar

Specify the Simulink model name. Generate code for the model before calling it for merge.

Example: `ModelName1`

a2lFilePath — Path and name of the A2L file

char vector | string scalar

Specify the path and name of the A2L file. If the A2L file is in the working directory,specify the file name.

Example: `folder/asap2FileName1.a2l`

Example: `asap2FileName2.a2l`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `MapFile='model.elf'` updates the ECU description object of the model by containing debug mapping information from the `model.elf` file.

Comments — Include comments in A2L file

`true` (default) | `false`

Generate the A2L file by including or excluding comments.

Example: `Comments=true`

CustomizationObject — Customize ASAP2 fields

`coder.asap2.UserCustomizeBase` object (default)

Create a user base and customize the ASAP2 fields such as:

- `ASAP2FileName`
- `ByteOrder`
- `HeaderComment`
- `ModParComment`
- `ModCommonComment`

Example: `CustomizationObject=obj`

CustomizeGroupsBy — Customize groups in ASAP2 file

`ARRAY` | `ASCII` | `SCALAR` | `MAP` | `CUBOID` | `CURVE` | `CUBE_4` | `CUBE_5`

Group the parameters and signals based on their properties.

Example: `CustomizeGroupsBy={'ARRAY', 'MAP'}`

EcuAddressExtension — Specify additional address information

`int`

Specify a 32-bit integer value to specify additional address information in the ASAP2 file. By default, no additional address information is required.

Example: `EcuAddressExtension=4`

FileName — Custom name for the exported A2L file

character vector | string scalar

Name for the exported A2L file.

Example: `FileName='test_car'`

Folder — Folder location to export A2L file

character vector | string scalar

Full path to a folder in which to place an exported A2L file.

Example: Folder='/home/temp/prjct/ '

GenerateXCPInfo — Include A2ML and IF_DATA in A2L file

true (default) | false

Generate the A2L file by including or excluding A2ML and IF_DATA sections.

Example: GenerateXCPInfo=true

IncludeAllRecordLayouts — Export record layouts of the base data types

false (default) | true

Export record layouts of all the base data types to a file named RecordLayouts.a2l according to the version of the A2L file.

Example: IncludeAllRecordLayouts=true

IncludeAutosarRteElements — Export RTE elements

true (default) | false

Generate the A2L file by including or excluding the AUTOSAR RTE elements.

Note This option is applicable only for AUTOSAR classic models.

Example: IncludeAutosarRteElements=true

IncludeDefaultEventList — Create default event

true (default) | false

Specify true to include a default event list in the A2L file.

Example: IncludeDefaultEventList=true

IncludeReferencedModels — Export referenced model elements

true (default) | false

Specify true to include referenced model elements in ASAP2 file. Specify false to exclude.

Example: IncludeReferencedModels=true

IncludeSharedElements — Export computation methods and record layouts separately

true (default) | false

Generate separate A2L files for computation methods and record layouts.

Note This option is applicable for models with ERT - based system target files.

Example: IncludeSharedElements=true

IndentFile — Follow indentation in A2L file

false (default) | true

Generate an A2L file by following indentation.

Example: IndentFile=true

MapFile — Name of symbol file for model

ELF | PDB | DWARF

Name of the model symbol file that contains symbols of generated code. For example, the addresses of variables used in generated code.

Example: MapFile='model.elf'

ModelClassName — Specify class instance and path names

character vector | string scalar

Custom model instance name in an A2L file. This argument is applicable only for AUTOSAR adaptive models.

Example: ModelClassName='customObj' or
ModelClassName='customNamespace::customObj'

Support64bitIntegers — Include or exclude the 64-bit integers in ASAP2 file

true (default) | false

Specify false to exclude 64-bit integers in the generated ASAP2 file.

Example: Support64bitIntegers=false

SupportStructureElements — Include or exclude the structure elements in ASAP2 file

true (default) | false

Specify false to exclude elements that are part of the structure in the generated ASAP2 file.

Example: SupportStructureElements=false

ToggleArrayLayout — Modify the array layout

false (default) | true

Update the array layout to ROW_DIR or COL_DIR to view the array data in row wise or column wise.

Example: ToggleArrayLayout=true

UseSavedSettings — Save and use the preferences

false (default) | true

Save the preferences made and generate the ASAP2 file with the preferences updated such as including the comments, version of the ASAP2 file.

Example: UseSavedSettings=false

Version — Version of A2L file

1.71 (default) | 1.31 | 1.61

A2L file format based on the ASAM MCD-2 MC standard defined by ASAM. There are multiple versions of the ASAM MCD-2 MC standard. Specify the version of A2L that you want.

Example: `Version='1.61'` or `Version='1.31'`

Output Arguments

mergeObj — Merged object

Custom ECU Description object

ECU description object containing the data elements from the specified Simulink model and A2L file. Use the object with `coder.asap2.export` function to export the merged A2L file.

Version History

Introduced in R2023a

See Also

`coder.asap2.export`

Topics

“Generate ASAP2 and CDF Calibration Files”

`coder.asap2.AxisInfo`

Set values for axis to describe lookup table

Description

Create an axis info object to define axis for a lookup table element and export it to an ASAP2 file.

Creation

`axis = coder.asap2.AxisInfo` creates an axis object. You can use the object to define axis for a lookup table element and export it to an ASAP2 file.

Properties

Name — Axis object name

character vector | string scalar

Specify a name for the custom axis object.

Example: "CustomLUT_1"

LongIdentifier — Axis object description

character vector | string scalar

Specify comments and description of the custom axis object.

Example: "Description of the LUT"

EcuAddress — ECU address

Hexadecimal

Address of the element in memory.

Example: 0x0000

RecordLayout — Record layout name

character vector | string scalar

Name of the record layout.

Example: "Record_DATATYPE"

AxisType — Type of axis

COM_AXIS | FIX_AXIS | STD_AXIS

Type of the axis.

Example: "FIX_AXIS"

InputQuantity — Input quantity

character vector | string scalar

Name of the measurement to be used as input quantity for the corresponding axis.

Example: 'NO_INPUT_QUANTITY'

CompuMethodName — Conversion method

character vector | string scalar

Name of the conversion method.

Example: "COMPU_METHOD_1"

MaxAxisPoints — Maximum axis points

integer

Maximum number of axis points.

Example: [4]

LowerLimit — Minimum value

numeric value

Possible minimum value.

Example: [1.5]

UpperLimit — Maximum value

numeric value

Possible maximum value.

Example: [11.5]

AxisPointsRef — Axis points reference

character vector | string scalar

Reference to the AXIS_PTS record for description of the axis points distribution.

Example: "AXIS_PTS_REF_1"

FixAxisType — Fix axis type

"FIX_AXIS_PAR" | "FIX_AXIS_PAR_DIST" | "FIX_AXIS_PAR_LIST"

Type of the fix axis.

Example: "FIX_AXIS_PAR"

Distance — Distance value

numerical value

Distance attribute for FIX_AXIS_PAR and FIX_AXIS_PAR_DIST.

Example: 2.3

Offset — Offset value

numerical value

Offset attribute for FIX_AXIS_PAR, FIX_AXIS_PAR_DIST, and list of values for FIX_AXIS_PAR_LIST.

Example: 0.3

Format — Display format

character vector | string scalar

Display format of the object in calibration tool.

Example: "%4.2"

DisplayIdentifier — Display name

character vector | string scalar

Display name of the object in calibration tool.

Example: "Custom name"

CalibrationAccess — Access type

Calibration | NoCalibration

Specify the calibration access type for the custom axis info object.

Example: "Calibration"

EcuAddressComment — ECU address comment

character vector | string scalar

Specify symbols for address replacement.

Example: "/* @ECU_Address@AXIS_1@ */"

EcuAddressExtension — ECU address extension

integer

Additional address information to distinguish different address spaces of an ECU (multi-micro controller devices).

Example: 1

SymbolLink — Reference to symbol

struct

Reference to symbol name within a linker map file.

Example: struct('SymbolName','p1_SymbolName','Offset',1)

MaxRefresh — Refresh rate

struct

Maximum refresh rate.

Example: struct('ScalingUnit',ScalingUnits.ONE_MILLI_SEC,'Rate',100)

CustomData — Additional description

character vector | string scalar

Specify any additional description that needs to be populated in the axis.

Example: "additional information"

Examples

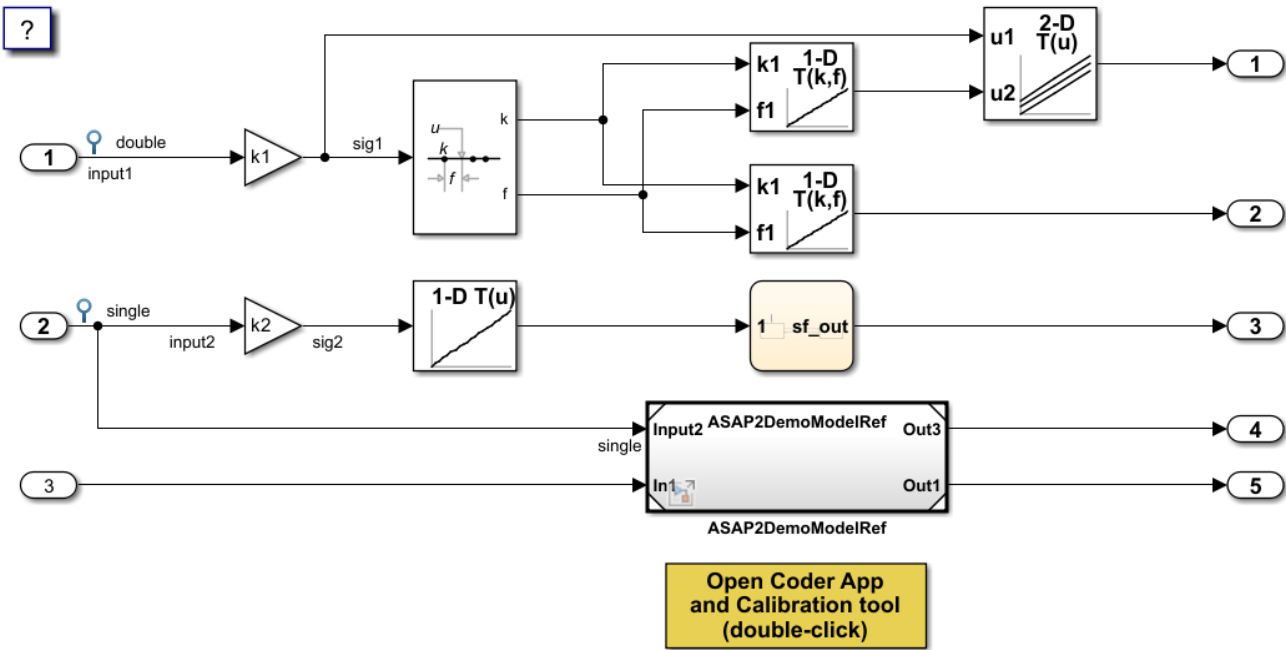
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model ASAP2Demo

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Build Time
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

```

Model          Build Reason          Status          Bu
=====
ASAP2Demo      Information cache folder or artifacts were missing. Code generated and compiled. 0h
2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

```

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

```
Export duration: 0h 0m 0s 165ms
```

Get the list of available computation methods in the description object.

```
find(descObj, "CompuMethod")
```

```
ans = 1x9 string
      "CM_double"      "CM_double_m_per_U0028_s_U005E_2_U0029"      "CM_double_rpm"      "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj, "CompuMethod", Units='rpm')
```

```
ans = 1x3 string
      "CM_double_rpm"      "CM_int16_rpm"      "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj, CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj, "CompuMethod", "CompuMethod_1")
```

```
ans =
    CompuMethod with properties:

        Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
        Format: '%2.3'
        Units: 's'
    Coefficients: [2 3]
    ConversionType: 'LINEAR'
    CompuVTabValues: [1x1 struct]
```



```
CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj, "CompuMethod", "CompuMethod_1", ConversionType="TAB_VERB");
set(descObj, "CompuMethod", "CompuMethod_1", CompuVTabValues = struct('Literals', ["false" "true"], 'Values', [0 1]));
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj, "CompuMethod", "CompuMethod_1")
```

```
modifiedprop =
    CompuMethod with properties:
        Name: 'CompuMethod_1'
        LongIdentifier: 'longIdentifierTest'
        Format: '%2.3'
        Units: 's'
        Coefficients: [2 3]
        ConversionType: "TAB_VERB"
        CompuVTabValues: [1x1 struct]
        CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:
    Literals: ["false" "true"]
    Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj, "CompuMethod", "CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj, Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj, "Characteristic", "Custom_parameter1")
```

```
ans =
```

```
Characteristic with properties:
```

```
        Name: 'Custom_parameter1'
    LongIdentifier: 'longIdentifierParam'
        Type: 'VALUE'
    EcuAddress: '0x0000'
    CompuMethodName: 'NO_COMPU_METHOD'
    LowerLimit: 0
    UpperLimit: 255
    EcuAddressComment: ""
    EcuAddressExtension: []
    CalibrationAccess: 'Calibration'
    DisplayIdentifier: ""
        Format: ""
        BitMask: []
        AxisInfo: []
    RecordLayout: ""
    Dimensions: []
    Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```

ans =
    Measurement with properties:
        Name: 'Custom_signal1'
        LongIdentifier: 'longIdentifierSignal'
        DataType: 'UBYTE'
        EcuAddress: '0x0000'
        CompuMethodName: ''
        LowerLimit: 0
        UpperLimit: 255
        Raster: [1x1 struct]
        EcuAddressComment: ''
        EcuAddressExtension: []
        CalibrationAccess: 'NoCalibration'
        DisplayIdentifier: ''
        Format: ''
        BitMask: []
        Dimensions: []
        Export: 1
        MaskData: [1x1 struct]
        MaxRefresh: [1x1 struct]
        SymbolLink: [1x1 struct]
        CustomData: ''

```

To modify a property of the measurement, use `set` function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```

LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";

```

Create and add axis information to the parameter.

```

axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;

```

Add the lookup table parameter to the description object.

```
add(descObj,LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Version History

Introduced in R2022b

See Also

`coder.asap2.Characteristic` | `coder.asap2.getEcuDescriptions` | `add`

Topics

"Generate ASAP2 and CDF Calibration Files"

coder.asap2.Characteristic

Create characteristic object

Description

Create a characteristic object to define parameter element and export it to an ASAP2 file.

Creation

`param = coder.asap2.Characteristic` creates a characteristic object. You can use the object to define a characteristic element and export it to an ASAP2 file.

Properties

Name — Parameter name

character vector | string scalar

Specify a name for the custom parameter.

Example: "CustomParameter_1"

LongIdentifier — Parameter description

character vector | string scalar

Specify comments and description of the parameter.

Example: "Description of the parameter"

Type — Type of parameter

VALUE (default) | ASCII | CURVE | MAP | CUBOID | CUBE_4 | CUBE_5 | VAL_BLK

Specify the list of parameters referred in the group which are already present in the description object.

Example: MAP

EcuAddress — Address of the parameter

hexadecimal value

Specify the address of the custom parameter.

Example: 0x1234

CompuMethodName — Computation method name

string scalar | character vector

Specify a name for the conversion method of the custom parameter.

Example: custom_conversionmethod

LowerLimit — Minimum value

parameter value

Specify the minimum possible value for the custom parameter.

Example: 0

UpperLimit — Maximum value

parameter value

Specify the maximum possible value for the custom parameter.

Example: 63365

EcuAddressComment — Address replacement

string scalar | character vector

Specify symbols for address replacement.

Example: `"/* @ECU_Address@CHARACTERISTIC_1@ */"`

EcuAddressExtension — Address information

integer

Specify additional address information.

Example: 1

CalibrationAccess — Access type

Calibration | NoCalibration | NotAccessible | ReadOnly | ReadWrite

Specify the calibration access type for the custom parameter.

Example: ReadOnly

DisplayIdentifier — Display name

string scalar | character vector

Specify a display name for the parameter to appear on calibration tool.

Example: CustomParam_1

Format — Display format

string scalar | character vector

Specify a display format for the parameter to appear on calibration tool.

Example: %3.4

BitMask — Bit handle

hexadecimal value

Specify to handle the single bits.

Example: 0x3a

AxisInfo — Axis description

coder.asap2.AxisInfo object

Specify the parameters for the axis description.

Example: `Custom_Axis`

ExportArrayAsFixAxis — Export as fix axis or not

false (default) | true

Specify `true` to export array as a look up table of type `FIX_AXIS`.

Example: `true`

Transpose — Transpose the axis

false (default) | true

Set `true` to transpose the axis. It is applicable only for lookup tables

Example: `true`

ComparisonQuantity — Reference variable

string scalar | character vector

Specify a valid measurement name as a reference in the ASAP2 file. It is applicable only for lookup tables.

Example: `Custom_Measurement`

RecordLayout — Record layout name

character vector | string scalar

Name of the record layout.

Example: `"Record_DATATYPE"`

Dimensions — Dimensions for array

numeric value

Specify the dimensions for an array data type.

Example: `2`

ForceShared — Standard axis as com axis

false (default) | true

Specify `true` to describe standard axis as com axis.

Example: `true`

Export — Export parameter

true (default) | false

Specify `true` to export the parameter and `false` to not export the parameter to ASAP2 file.

Example: `false`

SymbolLink — Symbol name and offset

struct

Specify reference to symbol name within a linker map file.

Example: `struct('SymbolName', 'p1_SymbolName', 'Offset', 1)`

MaxRefresh — Maximum refresh value

struct

Specify the maximum refresh rate.

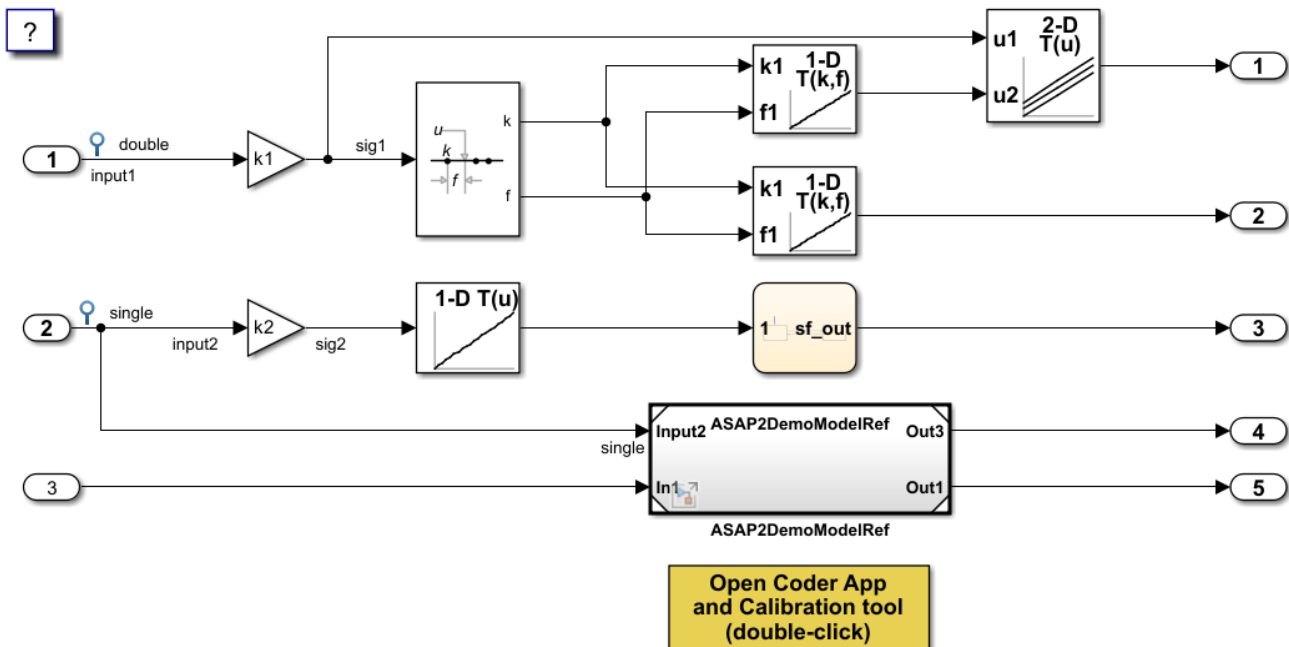
Example: `struct('ScalingUnit',ScalingUnits.ONE_MILLI_SEC,'Rate',100)`**CustomData — Additional description**

character vector | string scalar

Specify any additional description that needs to be populated in the characteristic.

Example: `"additional information"`**Examples****Customize an ASAP2 File**

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example ModelOpen the example model *ASAP2Demo*`open_system("ASAP2Demo");`

Copyright 1994-2022 The MathWorks, Inc.

Build the model.

`slbuild("ASAP2Demo");`


```
### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Build Time
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h 0m 0s 0ms

Top model targets:

Model	Build Reason	Status	Build Time
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h 0m 0s 0ms

2 of 2 models built (0 models already up to date)

Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Export duration: 0h 0m 0s 165ms

Get the list of available computation methods in the description object.

```
find(descObj, "CompuMethod")
```

```
ans = 1x9 string
    "CM_double"    "CM_double_m_per_U0028_s_U005E_2_U0029"    "CM_double_rpm"    "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj, "CompuMethod", Units='rpm')
```

```
ans = 1x3 string
    "CM_double_rpm"    "CM_int16_rpm"    "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj,CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj,"CompuMethod","CompuMethod_1")
```

```
ans =
  CompuMethod with properties:

      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
CompuVTabValues: [1x1 struct]
      CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj,"CompuMethod","CompuMethod_1",ConversionType="TAB_VERB");
set(descObj,"CompuMethod","CompuMethod_1",CompuVTabValues = struct('Literals',["false" "true"],'
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj,"CompuMethod","CompuMethod_1")
```

```
modifiedprop =
  CompuMethod with properties:

      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: "TAB_VERB"
CompuVTabValues: [1x1 struct]
      CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:
  Literals: ["false" "true"]
  Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj,"CompuMethod","CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj,Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj,"Characteristic","Custom_parameter1")
```

```
ans =
    Characteristic with properties:

        Name: 'Custom_parameter1'
    LongIdentifier: 'longIdentifierParam'
        Type: 'VALUE'
      EcuAddress: '0x0000'
    CompuMethodName: 'NO_COMPU_METHOD'
      LowerLimit: 0
      UpperLimit: 255
    EcuAddressComment: ""
    EcuAddressExtension: []
    CalibrationAccess: 'Calibration'
    DisplayIdentifier: ""
        Format: ""
      BitMask: []
      AxisInfo: []
    RecordLayout: ""
      Dimensions: []
        Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj,"Characteristic","Custom_parameter1",UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj,"Characteristic","Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;  
Signal_1.Name = 'Custom_signal1';  
Signal_1.LongIdentifier = 'longIdentifierSignal';  
Signal_1.UpperLimit = 255;  
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj,Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj,"Measurement","Custom_signal1")
```

```
ans =  
    Measurement with properties:  
  
                Name: 'Custom_signal1'  
    LongIdentifier: 'longIdentifierSignal'  
           DataType: 'UBYTE'  
        EcuAddress: '0x0000'  
    CompuMethodName: ''  
         LowerLimit: 0  
         UpperLimit: 255  
           Raster: [1x1 struct]  
    EcuAddressComment: ''  
    EcuAddressExtension: []  
    CalibrationAccess: 'NoCalibration'  
    DisplayIdentifier: ''  
           Format: ''  
          BitMask: []  
        Dimensions: []  
           Export: 1  
        MaskData: [1x1 struct]  
        MaxRefresh: [1x1 struct]  
        SymbolLink: [1x1 struct]  
        CustomData: ''
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj,"Measurement","Custom_signal1",CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj,"Measurement","Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj,LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Version History

Introduced in R2022b

See Also

[coder.asap2.AxisInfo](#) | [coder.asap2.Measurement](#) | [coder.asap2.getEcuDescriptions](#) | [add](#)

Topics

“Generate ASAP2 and CDF Calibration Files”

`coder.asap2.CompuMethod`

Create computation method object

Description

Create a computation method object to define computation method element and export it to an ASAP2 file.

Creation

`compu = coder.asap2.CompuMethod` creates a computation method object. You can use the object to define a computation method and export it to an ASAP2 file.

Properties

Name — computation method name

character vector | string scalar

Specify a name for the custom computation method.

Example: `"CustomCompuMethod_1"`

LongIdentifier — computation method description

character vector | string scalar

Specify comments and description of the computation method.

Example: `"Description of the computation method"`

Format — Display format

string scalar | character vector

Specify a display format for the computation method to appear on calibration tool.

Example: `%3.4`

Units — Physical units

string scalar | character vector

Specify the physical units for the computation method.

Example: `"sec"`

Coefficients — Fractional rational function coefficients

numeric

Specify the coefficients for the fractional rational function.

Example: `0 100 0 0 1 1`

ConversionType — Conversion method type

LINEAR | IDENTICAL | TAB_VERB | RAT_FUNC

Specify the conversion type for internal source value to physical value.

Example: LINEAR

CompuVTabValues — TAB_VERB values

struct

Specify the values for computation method with ConversionType as TAB_VERB.

Example: struct with literals and values

CustomData — Additional description

character vector | string scalar

Specify any additional description that needs to be populated in the computation method.

Example: "additional information"

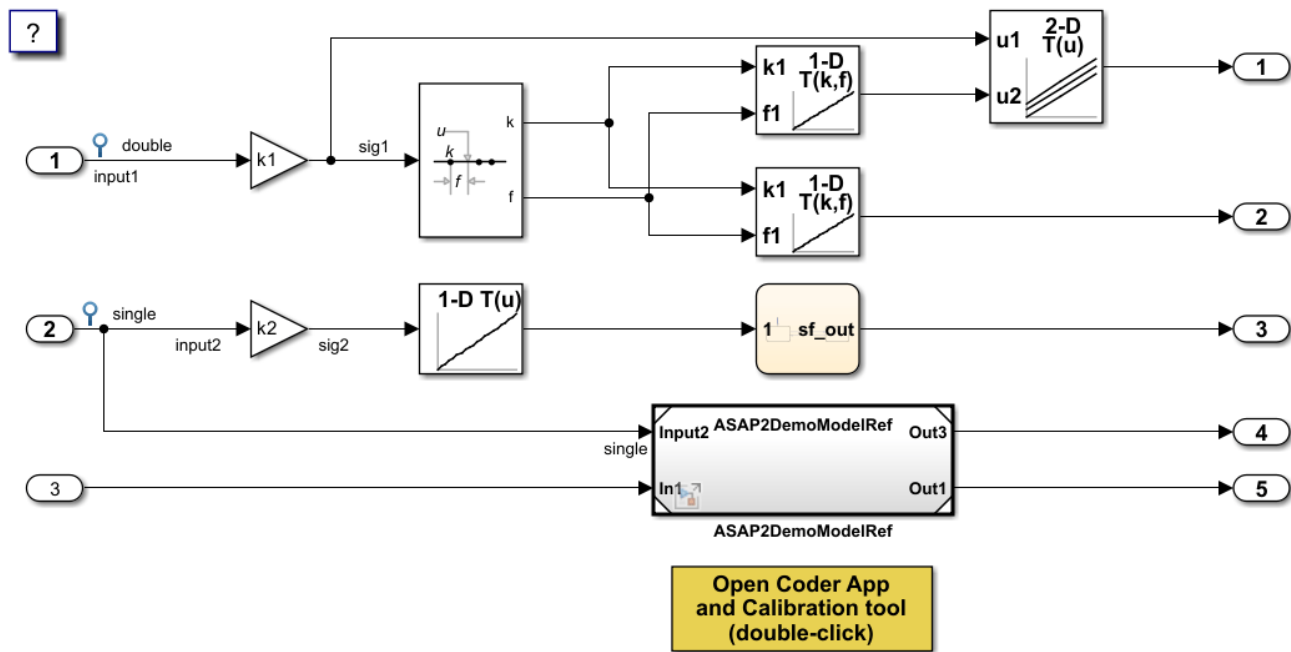
Examples**Customize an ASAP2 File**

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```


Export duration: 0h 0m 0s 165ms

Get the list of available computation methods in the description object.

```
find(descObj, "CompuMethod")
```

```
ans = 1x9 string
      "CM_double"      "CM_double_m_per_U0028_s_U005E_2_U0029"      "CM_double_rpm"      "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj, "CompuMethod", Units='rpm')
```

```
ans = 1x3 string
      "CM_double_rpm"      "CM_int16_rpm"      "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj, CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj, "CompuMethod", "CompuMethod_1")
```

```
ans =
  CompuMethod with properties:
      Name: 'CompuMethod_1'
 LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
 CompuVTabValues: [1x1 struct]
      CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj, "CompuMethod", "CompuMethod_1", ConversionType="TAB_VERB");
set(descObj, "CompuMethod", "CompuMethod_1", CompuVTabValues = struct('Literals', ["false" "true"], 'V'
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj, "CompuMethod", "CompuMethod_1")
```

```
modifiedprop =  
    CompuMethod with properties:  
  
        Name: 'CompuMethod_1'  
        LongIdentifier: 'longIdentifierTest'  
        Format: '%2.3'  
        Units: 's'  
        Coefficients: [2 3]  
        ConversionType: "TAB_VERB"  
        CompuVTabValues: [1x1 struct]  
        CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:  
    Literals: ["false"      "true"]  
    Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj,"CompuMethod","CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;  
Parameter_1.Name = 'Custom_parameter1';  
Parameter_1.LongIdentifier = 'longIdentifierParam';  
Parameter_1.UpperLimit = 255;  
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj,Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj,"Characteristic","Custom_parameter1")
```

```
ans =  
    Characteristic with properties:  
  
        Name: 'Custom_parameter1'  
        LongIdentifier: 'longIdentifierParam'  
        Type: 'VALUE'  
        EcuAddress: '0x0000'  
        CompuMethodName: 'NO_COMPU_METHOD'  
        LowerLimit: 0  
        UpperLimit: 255
```

```

    EcuAddressComment: ""
    EcuAddressExtension: []
    CalibrationAccess: 'Calibration'
    DisplayIdentifier: ""
        Format: ""
        BitMask: []
        AxisInfo: []
    RecordLayout: ""
        Dimensions: []
        Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""

```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```

Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;

```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```

ans =
    Measurement with properties:

        Name: 'Custom_signal1'
    LongIdentifier: 'longIdentifierSignal'
        DataType: 'UBYTE'
        EcuAddress: '0x0000'
    CompuMethodName: ''
        LowerLimit: 0
        UpperLimit: 255
        Raster: [1x1 struct]
    EcuAddressComment: ''

```

```
EcuAddressExtension: []
CalibrationAccess: 'NoCalibration'
DisplayIdentifier: ""
    Format: ""
    BitMask: []
Dimensions: []
    Export: 1
    MaskData: [1x1 struct]
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
CustomData: ""
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Version History

Introduced in R2022b

See Also

[coder.asap2.Characteristic](#) | [coder.asap2.Measurement](#) |
[coder.asap2.getEcuDescriptions](#) | [add](#)

Topics

“Generate ASAP2 and CDF Calibration Files”

coder.asap2.Function

Create function object

Description

Create a function object to define function description and export it to an ASAP2 file.

Creation

`userFunction = coder.asap2.Function` creates a function object. You can use the object to define new function description and export it to an ASAP2 file.

Properties

Name — Function name

character vector | string scalar

Specify a name for the function.

Example: `"CustomFunction_1"`

LongIdentifier — Function description

character vector | string scalar

Specify comments and description for the custom function.

Example: `"Description of the function"`

FunctionVersion — Function version

character vector | string scalar

Specify the version for the custom function.

Example: `"Custom_Function_version"`

Annotation — Notes

character vector | string scalar

Set of notes represented as multi-line ASCII description texts.

Example: `"Custom_Function_Annotation"`

InMeasurements — Input measurements

character vector | string scalar | array

Specify array of input measurements.

Example: `["measurement1", "measurement2"]`

OutMeasurements — Output measurements

character vector | string scalar | array

Specify array of output measurements.

Example: ["measurement1", "measurement2"]

LocMeasurements — Local measurements

character vector | string scalar | array

Specify array of local measurements.

Example: ["measurement1", "measurement2"]

DefCharacteristics — Defined characteristics

character vector | string scalar | array

Specify array of defined characteristics.

Example: ["characteristic1", "characteristic2"]

RefCharacteristics — Referenced characteristics

character vector | string scalar | array

Specify array of referenced characteristics.

Example: ["characteristic1", "characteristic2"]

SubFunctions — Sub functions

character vector | string scalar | array

Specify array of sub functions.

Example: ["function1", "function2"]

Examples

Create and Add Custom Function

This example shows how to add a custom function to the ASAP2 file.

Create ECU Description Object

Open a model, build, and create ECU description object.

```
open_system("ASAP2Demo")
slbuild("ASAP2Demo")

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Build

```
=====
ASAP2DemoModelRef Target (ASAP2DemoModelRef.c) did not exist. Code generated and compiled. 0h
```

```
Top model targets:
```

Model	Build Reason	Status	Build Duration
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

```
2 of 2 models built (0 models already up to date)
Build duration: 0h 1m 8.4627s
```

```
descObj=coder.asap2.getEcuDescriptions("ASAP2Demo")
```

```
Export duration: 0h 0m 0s 363ms
```

```
descObj =
    Data with no properties.
```

Create Custom Function

Create a function object template and add the necessary fields.

```
userFunction = coder.asap2.Function
```

```
userFunction =
    Function with properties:
        Name: "undefined"
        FunctionVersion: [1x0 char]
        LongIdentifier: [1x0 char]
        Annotation: [1x0 char]
        InMeasurements: [1x0 string]
        OutMeasurements: [1x0 string]
        LocMeasurements: [1x0 string]
        DefCharacteristics: [1x0 string]
        RefCharacteristics: [1x0 string]
        SubFunctions: [1x0 string]
```

```
userFunction.Name = 'Custom_Function1'
```

```
userFunction =
    Function with properties:
        Name: 'Custom_Function1'
        FunctionVersion: [1x0 char]
        LongIdentifier: [1x0 char]
        Annotation: [1x0 char]
        InMeasurements: [1x0 string]
        OutMeasurements: [1x0 string]
        LocMeasurements: [1x0 string]
        DefCharacteristics: [1x0 string]
        RefCharacteristics: [1x0 string]
        SubFunctions: [1x0 string]
```

```
userFunction.LongIdentifier = 'Custom Function by user'
```



```

userFunction =
    Function with properties:

        Name: 'Custom_Function1'
        FunctionVersion: [1x0 char]
        LongIdentifier: 'Custom Function by user'
        Annotation: [1x0 char]
        InMeasurements: [1x0 string]
        OutMeasurements: [1x0 string]
        LocMeasurements: [1x0 string]
        DefCharacteristics: [1x0 string]
        RefCharacteristics: [1x0 string]
        SubFunctions: [1x0 string]

```

Export the Function to ASAP2 File

Add the function to ECU description object and generate ASAP2 file using the description object.

```

add(descObj,userFunction)
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj)

```

Version History

Introduced in R2022b

See Also

add | coder.asap2.getEcuDescriptions

Topics

“Generate ASAP2 and CDF Calibration Files”

`coder.asap2.Measurement`

Create measurement object

Description

Create a measurement object to define a signal and export it to an ASAP2 file.

Creation

`signal = coder.asap2.Measurement` creates a measurement object. You can use the object to define a signal and export it to an ASAP2 file.

Properties

Name — Variable name

character vector | string scalar

Specify a name for the custom variable.

Example: "CustomVariable_1"

LongIdentifier — Variable description

character vector | string scalar

Specify comments and description of the variable.

Example: "Description of the variable"

DataType — Variable data type

character vector | string scalar

Specify a data type for the variable.

Example: UBYTE

EcuAddress — Address of the variable

hexadecimal value

Specify the address of the custom variable.

Example: 0x1234

CompuMethodName — Computation method name

string scalar | character vector

Specify a name for the conversion method of the custom variable.

Example: custom_conversionmethod

LowerLimit — Minimum value

parameter value

Specify the minimum possible value for the custom variable.

Example: 0

UpperLimit — Maximum value

parameter value

Specify the maximum possible value for the custom variable.

Example: 65535

EcuAddressComment — Address replacement

string scalar | character vector

Specify symbols for address replacement.

Example: `"/* @ECU_Address@MEASUREMENT_1@ */"`

EcuAddressExtension — Address information

integer

Specify additional address information.

Example: 1

CalibrationAccess — Access type

Calibration | NoCalibration | NotAccessible | ReadOnly | ReadWrite

Specify the calibration access type for the custom variable.

Example: ReadOnly

DisplayIdentifier — Display name

string scalar | character vector

Specify a display name for the variable to appear on calibration tool.

Example: CustomVariable_1

Format — Display format

string scalar | character vector

Specify a display format for the variable to appear on calibration tool.

Example: %3.4

BitMask — Bit handle

hexadecimal value

Specify to handle the single bits.

Example: 0x3a

Dimensions — Dimensions for array

numeric value

Specify the dimensions for an array data type.

Example: 2

Export — Export variable

true | false

Specify `true` to export the variable and `false` to not export the variable to ASAP2 file.

Example: `false`

SymbolLink — Symbol name and offset

struct with name and value

Specify reference to symbol name within a linker map file.

Example: `struct('SymbolName','p1_SymbolName','Offset',1)`

MaskData — Mask data

struct

Perform operation on masked out value. This property can be used with the `BitMask`.

Example: `struct('Shift',3,'SignExtend',1)`

MaxRefresh — Maximum refresh value

parameter value

Specify the maximum refresh rate.

Example: `struct('ScalingUnit',ScalingUnits.ONE_MILLI_SEC,'Rate',100)`

CustomData — Additional description

character vector | string scalar

Specify any additional description that needs to be populated in the measurement.

Example: `"additional information"`

Examples

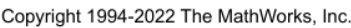
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



```
slbuild("ASAP2Demo");  
### Searching for referenced models in model 'ASAP2Demo'.  
### Total of 2 models to build.  
### Starting serial code generation build.  
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef  
### Starting build procedure for: ASAP2Demo  
### Successful completion of build procedure for: ASAP2Demo
```

Model reference code generation targets:

Top model targets:

```
2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s
```

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

```
Export duration: 0h 0m 0s 165ms
```

Get the list of available computation methods in the description object.

```
find(descObj, "CompuMethod")
```

```
ans = 1x9 string
      "CM_double"      "CM_double_m_per_U0028_s_U005E_2_U0029"      "CM_double_rpm"      "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj, "CompuMethod", Units='rpm')
```

```
ans = 1x3 string
      "CM_double_rpm"      "CM_int16_rpm"      "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj, CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj, "CompuMethod", "CompuMethod_1")
```

```
ans =
  CompuMethod with properties:
      Name: 'CompuMethod_1'
 LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
 CompuVTabValues: [1x1 struct]
      CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj, "CompuMethod", "CompuMethod_1", ConversionType="TAB_VERB");
set(descObj, "CompuMethod", "CompuMethod_1", CompuVTabValues = struct('Literals', ["false" "true"], 'V'
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj, "CompuMethod", "CompuMethod_1")
```

```

modifiedprop =
    CompuMethod with properties:

        Name: 'CompuMethod_1'
        LongIdentifier: 'longIdentifierTest'
        Format: '%2.3'
        Units: 's'
        Coefficients: [2 3]
        ConversionType: "TAB_VERB"
        CompuVTabValues: [1x1 struct]
        CustomData: ""

```

```
modifiedprop.CompuVTabValues
```

```

ans = struct with fields:
    Literals: ["false"      "true"]
    Values: [0 1]

```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj,"CompuMethod","CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```

Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;

```

Add the custom characteristic to the ECU description object.

```
add(descObj,Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj,"Characteristic","Custom_parameter1")
```

```

ans =
    Characteristic with properties:

        Name: 'Custom_parameter1'
        LongIdentifier: 'longIdentifierParam'
        Type: 'VALUE'
        EcuAddress: '0x0000'
        CompuMethodName: 'NO_COMPU_METHOD'
        LowerLimit: 0
        UpperLimit: 255

```

```
EcuAddressComment: ""
EcuAddressExtension: []
CalibrationAccess: 'Calibration'
DisplayIdentifier: ""
    Format: ""
    BitMask: []
    AxisInfo: []
RecordLayout: ""
    Dimensions: []
    Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```
ans =
```

```
Measurement with properties:
```

```
    Name: 'Custom_signal1'
LongIdentifier: 'longIdentifierSignal'
    DataType: 'UBYTE'
    EcuAddress: '0x0000'
CompuMethodName: ""
    LowerLimit: 0
    UpperLimit: 255
    Raster: [1x1 struct]
EcuAddressComment: ""
```



```

EcuAddressExtension: []
CalibrationAccess: 'NoCalibration'
DisplayIdentifier: ""
    Format: ""
    BitMask: []
Dimensions: []
    Export: 1
    MaskData: [1x1 struct]
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
CustomData: ""

```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```

LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";

```

Create and add axis information to the parameter.

```

axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;

```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Version History

Introduced in R2022b

See Also

`coder.asap2.Characteristic` | `coder.asap2.getEcuDescriptions` | `add`

Topics

“Generate ASAP2 and CDF Calibration Files”

coder.asap2.Group

Create group object

Description

Create a group object to view characteristics and measurements together and export it to an ASAP2 file.

Creation

`groupObj = coder.asap2.Group` creates a group object. You can use the object to define list of characteristic, measurements, subgroups to the group and export it to an ASAP2 file.

Properties

Name — Group name

character vector | string scalar

Specify a name for the custom group.

Example: "CustomGroup_1"

LongIdentifier — Group description

character vector | string scalar

Specify comments and description of the group.

Example: "Description of the group"

RefCharacteristic — Array of characteristics

array of parameter names

Specify the list of parameters referred in the group.

Example: ["Parameter1", "Parameter2"]

RefMeasurement — Array of measurements

array of signal names

Specify the list of signals referred in the group.

Example: ["Signal1", "Signal2"]

Root — Group hierarchy

true | false

Specifies the root of the group's hierarchy.

Example: 1

SubGroup — Array of groups

array of group names

Specify the list of sub groups referred in the group.

Example: ["Subgroup1", "Subgroup2"]

CustomData — Additional description

character vector | string scalar

Specify any additional description that needs to be populated in the group.

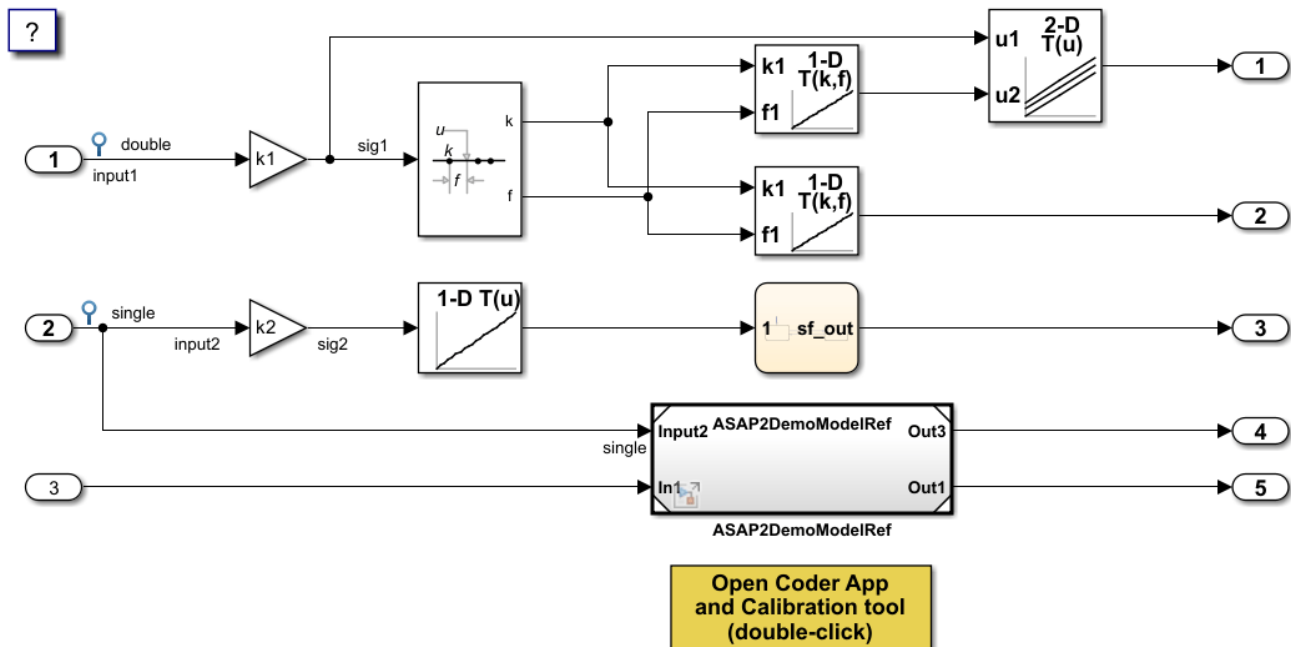
Example: "additional information"

Examples**Create and Add Custom Group**

Add, update, filter, find, and remove ECU descriptions for groups in an ASAP2 file.

Open and Build Example ModelOpen the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");
```

```
### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)

Build duration: 0h 0m 29.879s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Export duration: 0h 0m 0s 188ms

Get the list of available groups in the description object.

```
find(descObj, "Group")
```

```
ans = 1x2 string
    "ASAP2Demo"    "ASAP2Demo.ASAP2DemoModelRef.ASAP2DemoModelRef"
```

Filter the groups and get a list of groups that has Root set to true.

```
find(descObj, "Group", Root=true)
```

```
ans =
    "ASAP2Demo"
```

Create and Add Custom Group

To add a new group to the ASAP2 file, create a custom group.

```
GroupObj_GR1 = coder.asap2.Group;
GroupObj_GR1.Name = 'CustomGroup_1';
GroupObj_GR1.LongIdentifier = 'New test group';
GroupObj_GR1.RefCharacteristic = ["ydata3", "ydata4"];
GroupObj_GR1.RefMeasurement = ["ASAP2Demo_Y.Out3", "ASAP2Demo_Y.Out3"];
GroupObj_GR1.Root = true;
```

Add the custom group to the ECU description object.

```
add(descObj, GroupObj_GR1);
```

Get Group Properties

Get the properties of newly added group by using this command.

```
get(descObj, "Group", "CustomGroup_1")

ans =
    Group with properties:

        Name: 'CustomGroup_1'
    LongIdentifier: 'New test group'
  RefCharacteristic: ["ydata3"      "ydata4"]
    RefMeasurement: ["ASAP2Demo_Y.Out3"      "ASAP2Demo_Y.Out3"]
        Root: 1
      SubGroup: [1x0 string]
    CustomData: [1x0 string]
```

Update Group Properties

To modify a property of the group, use *set* function. Update the *LongIdentifier* field of the group.

```
set(descObj, "Group", "CustomGroup_1", LongIdentifier="Group 1 new long identifier")
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the group CustomGroup_1.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Delete Group

Remove the newly added group from the description object.

```
delete(descObj, "Group", "CustomGroup_1");
```

Version History

Introduced in R2023b

See Also

`coder.asap2.Measurement` | `coder.asap2.getEcuDescriptions` | `add`

Topics

“Generate ASAP2 and CDF Calibration Files”

“Create and Add Custom Data Elements”

“Customize Using Base Files”

coder.asap2.RecordLayout

Create record layout object

Description

Create a record layout object to define a record layout element and export it to an ASAP2 file.

Creation

`rec = coder.asap2.RecordLayout` creates a record layout object. You can use the object to define a record layout and export it to an ASAP2 file.

Properties

Name — Record layout name

character vector | string scalar

Specify a name for the custom record layout.

Example: "CustomRecordLayout_1"

Record — Record layout description

structure

Specify details for the record layout such as `Position`, `DataType`, `IndexMode`, `IndexOrder`, and `AddressType`.

Example:

```
struct('Name','Record1','Position',1,'DataType','int','IndexMode','COLUMN_DIR',  
'IndexOrder','INDEX_INCR','AddressType','DIRECT')
```

CustomData — Additional description

character vector | string scalar

Specify any additional description that needs to be populated in the record layout.

Example: "additional information"

Examples

Create a Custom Record Layout Object

Open a model, build it, and create the ECU description object.

```
openExample("ASAP2Demo")  
slbuild("ASAP2Demo")  
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo")
```

Create a custom record layout and add the necessary details.

```
userRecordLayout = coder.asap2.RecordLayout  
userRecordLayout.Name = 'Custom_RecordLayout1'
```

Add and export the custom record layout.

```
add(descObj,userRecordLayout)  
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj)
```

Version History

Introduced in R2023a

See Also

`add` | `coder.asap2.getEcuDescriptions`

Topics

"Generate ASAP2 and CDF Calibration Files"

coder.asap2.UserCustomizeBase

Customize project and module settings

Description

Create a custom user object to define project and module information in the ASAP2 file.

Creation

`Obj = coder.asap2.UserCustomizeBase` creates an object to define PROJECT, MODULE, MOD, MOD_PAR, and MOD_COMMON settings. Use the object to add the fields to ASAP2 file.

Properties

HeaderComment — Header information

Header Comments go here (default) | character vector

Specify header information on a project as a character vector.

ModParComment — Management data

MOD PAR Comment Goes Here (default) | character vector

Specify the description relating to the ECU-specific management data as a character vector.

ModCommonComment — Module data

Mod Common Comment Here (default) | character vector

Specify the general description data for the module, which are used as standard in this module as a character vector.

ASAP2FileName — ASAP2 file name

[] (default) | character vector

Specify the ASAP2 file name as a character vector.

ByteOrder — Byte order

[] (default) | MSB_FIRST | MSB_LAST

Specify the byte order of the relevant quantity.

ProjectName — Project identifier

ProjectName (default) | character vector

Specify a name for the project as a character vector.

ProjectComment — Project comment

Project Comment Goes Here (default) | character vector

Specify a comment for the project as a character vector.

ModuleName — Module name`ModuleName` (default) | character vector

Specify a name for the module as a character vector.

ModuleComment — Module comment`Module Comment Goes Here` (default) | character vector

Specify a comment for the module as a character vector.

AfterBeginHeaderContents — Header contents`[ ]` (default) | character vector

Specify the header contents after the header name as a character vector.

BeforeEndHeaderContents — Header contents`[ ]` (default) | character vector

Specify the header contents before ending the header as a character vector.

AfterBeginProjectContents — Project contents`[ ]` (default) | character vector

Specify the project contents after the project name as a character vector.

BeforeEndProjectContents — Project contents`[ ]` (default) | character vector

Specify the project contents before ending the project as a character vector.

AfterBeginModuleContents — Module contents`[ ]` (default) | character vector

Specify the module contents after the module name as a character vector.

BeforeEndModuleContents — Module contents`[ ]` (default) | character vector

Specify the module contents before ending the module as a character vector.

AfterBeginModParContents — Mod par contents`[ ]` (default) | character vector

Specify the mod par contents at the beginning of mod par as a character vector.

BeforeEndModParContents — Mod par contents`[ ]` (default) | character vector

Specify the mod par contents before ending of mod par as a character vector.

AfterBeginModCommonContents — Mod common contents`[ ]` (default) | character vector

Specify the mod common contents at the beginning of mod common as a character vector.

BeforeEndModCommonContents — Mod common contents

[] (default) | character vector

Specify the mod common contents before ending of mod common as a character vector.

AddA2MLSection — Interface specific data description

[] (default) | character vector

Specify the description for interface specific data as a character vector.

AddIFDataSection — Interface specific data

[] (default) | character vector

specify the interface specific data as a character vector.

Object Functions

writeFileHead	Get or set text at the beginning of ASAP2 file
writeFileTail	Get or set text at the end of ASAP2 file
writeHardwareInterface	Get or set custom data to describe ECU interface data
writeHeader	Get or set text from ASAP2 file header section

Examples

Define Custom Project and Module Information

This example shows how to add custom project and module information to an ASAP2 file.

Open and build the example model

Open the example model "ASAP2Demo"

```
open_system("ASAP2Demo");
slbuild("ASAP2Demo");
```

Create custom base object

Create a custom base object, define the project and module information.

```
Obj = coder.asap2.UserCustomizeBase;
Obj.ProjectName = 'Demo_project_name';
Obj.ProjectComment = 'Custom comment for the demo project';
Obj.ModuleName = 'Demo_module_name';
Obj.ModuleComment = 'Custom comment for the demo module';
Obj.HeaderComment = 'Custom comment';
```

Generate the ASAP2 file by passing the object using the command

```
coder.asap2.export("ASAP2Demo",CustomizationObject=Obj);
```

Version History

Introduced in R2022b

See Also

`coder.asap2.export`

Topics

“Generate ASAP2 and CDF Calibration Files”

writeFileHead

Get or set text at the beginning of ASAP2 file

Syntax

```
fileHeader = writeFileHead(UserCustomizeBaseObj)
```

Description

`fileHeader = writeFileHead(UserCustomizeBaseObj)` returns the text at the beginning of the ASAP2 file. To add a custom text, derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to update the `writeFileHead` function by adding custom text.

Examples

Add Custom Text to ASAP2 File

This example shows how to add custom text to the ASAP2 file.

To add custom data, you need to derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to pass the custom data such as header comments, text at the beginning and ending of the ASAP2 file, and description about ECU interface specific data.

This example uses ASAP2Demo model.

Build the Model

Open the model and build it.

```
open_system("ASAP2Demo");
slbuild("ASAP2Demo");
```

Create Derived Object

To create a derived object from `coder.asap2.UserCustomizeBase` class and reuse the functions by adding custom data, copy the below code and save it in a m-file.

```
% classdef TestASAP2Customization < coder.asap2.UserCustomizeBase
%     % Customization class for asap2 file
%     methods
%         function header = writeHeader(obj)
%             header = sprintf([' /begin HEADER ' 'test Header text' '\n',...
%                 'header test comments from user\n',...
%                 ' /end HEADER']);
%         end
%
%         function hardwareInterface = writeHardwareInterface(obj)
%             hardwareInterface = sprintf([' /begin MOD_PAR "' 'UserDefined' '"\n',...
%                 ' test comments from user\n',...
%                 ' /end MOD_PAR\n\n',...
%                 ' /begin MOD_COMMON "' 'UserDefined MOD_COMMON' '"\n',...
%                 ' /end MOD_COMMON']);
%         end
%     end
end
```

```
%           '      user defined values for MOD_COMMON\n',...
%           '      obj.ByteOrder '\n',...
%           '      /end MOD_COMMON']);
%
%   end
%
%   function fileTail = writeFileTail(obj)
%       fileTail = sprintf(['//* This File can be used for ' ,...
%       'calibration .*/\n'...
%       '/* EOF ' ,obj.ASAP2FileName, '*/']);
%   end
%
%   function fileHeader = writeFileHead(obj)
%       fileHeader = sprintf(['/*****\n',...
%       ' This is an a2l file which can be used for calibration ',...
%       ' with INCA or CANAPE\n',...
%       ' *****/'
%       ]);
%   end
% end
% end
```

Run the m-file to create a derived object and add the custom data to the object. Run the TestASAP2Customization m-file from the example folder that contains the class definition for the derived object.

```
obj = TestASAP2Customization;
```

Generate ASAP2 File

Generate the ASAP2 file for the model by passing the object created as a customization object.

```
coder.asap2.export('ASAP2Demo', CustomizationObject = obj);
```

An ASAP2 file generates by adding the custom data provided in the derived object.

Input Arguments

UserCustomizeBaseObj — Derived object name

character vector | string scalar

Specify the name of the object that is derived from `coder.asap2.UserCustomizeBase` class.

Example: `CustomizationObject`

Version History

Introduced in R2025a

See Also

`coder.asap2.UserCustomizeBase` | `coder.asap2.export` | `writeFileTail` | `writeHardwareInterface` | `writeHeader`

Topics

“Customize Using Base Files”

Get or set text at the end of ASAP2 file

```
fileTail = writeFileTail(UserCustomizeBaseObj)
```

`fileTail = writeFileTail(UserCustomizeBaseObj)` returns the text at the end of the ASAP2 file. To add a custom text, derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to update the `writeFileTail` function by adding custom text.

Add Custom Text to ASAP2 File

This example shows how to add custom text to the ASAP2 file.

To add custom data, you need to derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to pass the custom data such as header comments, text at the beginning and ending of the ASAP2 file, and description about ECU interface specific data.

This example uses ASAP2Demo model.

Open the model and build it.

```
open_system("ASAP2Demo");
slbuild("ASAP2Demo");
```

Create Derived Object

To create a derived object from `coder.asap2.UserCustomizeBase` class and reuse the functions by adding custom data, copy the below code and save it in a m-file.

```
% classdef TestASAP2Customization < coder.asap2.UserCustomizeBase
%     % Customization class for asap2 file
%     methods
%         function header = writeHeader(obj)
%             header = sprintf([' /begin HEADER ' 'test Header text' '\n',...
%                 'header test comments from user\n',...
%                 ' /end HEADER']);
%     end
%
%     function hardwareInterface = writeHardwareInterface(obj)
%         hardwareInterface = sprintf([' /begin MOD_PAR "' 'UserDefined' '"\n',...
%             ' test comments from user\n',...
%             ' /end MOD_PAR\n\n',...
%             ' /begin MOD COMMON "' 'UserDefined MOD COMMON' '"\n',...
```

```
%           '      user defined values for MOD_COMMON\n',...
%           ' obj.ByteOrder '\n',...
%           ' /end MOD_COMMON']);
%
%       end
%
%       function fileTail = writeFileTail(obj)
%           fileTail = sprintf(['//* This File can be used for ' ,...
%               'calibration .*/\n'...
%               '/* EOF ',obj.ASAP2FileName, '*/']);
%       end
%
%       function fileHeader = writeFileHead(obj)
%           fileHeader = sprintf(['/*****\n',...
%               ' This is an a2l file which can be used for calibration ',...
%               ' with INCA or CANAPE\n',...
%               ' *****/']);
%       end
%   end
% end
```

Run the m-file to create a derived object and add the custom data to the object. Run the TestASAP2Customization m-file from the example folder that contains the class definition for the derived object.

```
obj = TestASAP2Customization;
```

Generate ASAP2 File

Generate the ASAP2 file for the model by passing the object created as a customization object.

```
coder.asap2.export('ASAP2Demo', CustomizationObject = obj);
```

An ASAP2 file generates by adding the custom data provided in the derived object.

Input Arguments

UserCustomizeBaseObj — Derived object name

character vector | string scalar

Specify the name of the object that is derived from `coder.asap2.UserCustomizeBase` class.

Example: `CustomizationObject`

Version History

Introduced in R2025a

See Also

`coder.asap2.UserCustomizeBase` | `coder.asap2.export` | `writeFileHead` | `writeHardwareInterface` | `writeHeader`

Topics

“Customize Using Base Files”

writeHardwareInterface

Get or set custom data to describe ECU interface data

Syntax

```
hardwareInterface = writeHardwareInterface(UserCustomizeBaseObj)
```

Description

`hardwareInterface = writeHardwareInterface(UserCustomizeBaseObj)` returns the text that describes ECU interface specific data in the ASAP2 file. To add a custom text, derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to update the `writeHardwareInterface` function by adding custom text.

Examples

Add Custom Text to ASAP2 File

This example shows how to add custom text to the ASAP2 file.

To add custom data, you need to derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to pass the custom data such as header comments, text at the beginning and ending of the ASAP2 file, and description about ECU interface specific data.

This example uses ASAP2Demo model.

Build the Model

Open the model and build it.

```
open_system("ASAP2Demo");
slbuild("ASAP2Demo");
```

Create Derived Object

To create a derived object from `coder.asap2.UserCustomizeBase` class and reuse the functions by adding custom data, copy the below code and save it in a m-file.

```
% classdef TestASAP2Customization < coder.asap2.UserCustomizeBase
%     % Customization class for asap2 file
%     methods
%         function header = writeHeader(obj)
%             header = sprintf([' /begin HEADER ' 'test Header text' '\n',...
%                 'header test comments from user\n',...
%                 ' /end HEADER']);
%         end
%
%         function hardwareInterface = writeHardwareInterface(obj)
%             hardwareInterface = sprintf([' /begin MOD_PAR " ' 'UserDefined' '"\n',...
%                 ' test comments from user\n',...
%                 ' /end MOD_PAR\n\n',...
%
```

```
%
%         '    /begin MOD_COMMON "" 'UserDefined MOD_COMMON' ""\n',...
%         '    user defined values for MOD_COMMON\n',...
%         '    obj.ByteOrder '\n',...
%         '    /end MOD_COMMON']);
%
%     end
%
%     function fileTail = writeFileTail(obj)
%         fileTail = sprintf(['/* This File can be used for ' ,...
%             'calibration .*/\n'...
%             '/* EOF ',obj.ASAP2FileName, '*/']);
%
%     end
%
%     function fileHeader = writeFileHead(obj)
%         fileHeader = sprintf(['/******\n',...
%             ' This is an a2l file which can be used for calibration ',...
%             ' with INCA or CANAPE\n',...
%             ' *****/']);
%
%     end
%
% end
% end
```

Run the m-file to create a derived object and add the custom data to the object. Run the TestASAP2Customization m-file from the example folder that contains the class definition for the derived object.

```
obj = TestASAP2Customization;
```

Generate ASAP2 File

Generate the ASAP2 file for the model by passing the object created as a customization object.

```
coder.asap2.export('ASAP2Demo', CustomizationObject = obj);
```

An ASAP2 file generates by adding the custom data provided in the derived object.

Input Arguments

UserCustomizeBaseObj — Derived object name

character vector | string scalar

Specify the name of the object that is derived from `coder.asap2.UserCustomizeBase` class.

Example: `CustomizationObject`

Version History

Introduced in R2025a

See Also

`coder.asap2.UserCustomizeBase` | `coder.asap2.export` | `writeFileHead` | `writeFileTail` | `writeHeader`

Topics

“Customize Using Base Files”

writeHeader

Get or set text from ASAP2 file header section

Syntax

```
header = writeHeader(UserCustomizeBaseObj)
```

Description

`header = writeHeader(UserCustomizeBaseObj)` returns the text from the header section of the ASAP2 file. To add a custom text, derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to update the `writeHeader` function by adding custom text.

Examples

Add Custom Text to ASAP2 File

This example shows how to add custom text to the ASAP2 file.

To add custom data, you need to derive an object from the `coder.asap2.UserCustomizeBase` class. Use the derived object to pass the custom data such as header comments, text at the beginning and ending of the ASAP2 file, and description about ECU interface specific data.

This example uses ASAP2Demo model.

Build the Model

Open the model and build it.

```
open_system("ASAP2Demo");
slbuild("ASAP2Demo");
```

Create Derived Object

To create a derived object from `coder.asap2.UserCustomizeBase` class and reuse the functions by adding custom data, copy the below code and save it in a m-file.

```
% classdef TestASAP2Customization < coder.asap2.UserCustomizeBase
%     % Customization class for asap2 file
%     methods
%         function header = writeHeader(obj)
%             header = sprintf([' /begin HEADER ' 'test Header text' '\n',...
%                 'header test comments from user\n',...
%                 ' /end HEADER']);
%         end
%
%         function hardwareInterface = writeHardwareInterface(obj)
%             hardwareInterface = sprintf([' /begin MOD_PAR "' 'UserDefined' '"\n',...
%                 ' test comments from user\n',...
%                 ' /end MOD_PAR\n\n',...
%                 ' /begin MOD_COMMON "' 'UserDefined MOD_COMMON' '"\n',...
%                 ' /end MOD_COMMON']);
%         end
%     end
end
```

```
%           '      user defined values for MOD_COMMON\n',...
%           '      obj.ByteOrder '\n',...
%           '      /end MOD_COMMON']);
%
%   end
%
%   function fileTail = writeFileTail(obj)
%       fileTail = sprintf(['//* This File can be used for ' ,...
%       'calibration .*/\n'...
%       '/* EOF ',obj.ASAP2FileName, '*/']);
%   end
%
%   function fileHeader = writeFileHead(obj)
%       fileHeader = sprintf(['/*****\n',...
%       ' This is an a2l file which can be used for calibration ',...
%       ' with INCA or CANAPE\n',...
%       ' *****/'
%       ]);
%   end
% end
% end
```

Run the m-file to create a derived object and add the custom data to the object. Run the TestASAP2Customization m-file from the example folder that contains the class definition for the derived object.

```
obj = TestASAP2Customization;
```

Generate ASAP2 File

Generate the ASAP2 file for the model by passing the object created as a customization object.

```
coder.asap2.export('ASAP2Demo', CustomizationObject = obj);
```

An ASAP2 file generates by adding the custom data provided in the derived object.

Input Arguments

UserCustomizeBaseObj — Derived object name

character vector | string scalar

Specify the name of the object that is derived from `coder.asap2.UserCustomizeBase` class.

Example: `CustomizationObject`

Version History

Introduced in R2025a

See Also

`coder.asap2.UserCustomizeBase` | `coder.asap2.export` | `writeFileHead` | `writeFileTail` | `writeHardwareInterface`

Topics

“Customize Using Base Files”

coder.asap2.VarCharacteristic

Create variant parameter

Description

Create a variant characteristic object to define a custom variant parameter and export it to the ASAP2 file.

Creation

Syntax

```
var_param = coder.asap2.VarCharacteristic
```

Description

`var_param = coder.asap2.VarCharacteristic` creates a variant characteristic object. You can use the object to define a custom variant parameter and export it to the ASAP2 file.

Properties

Name — Variant parameter name

character vector | string scalar

Custom variant parameter name, specified as a character vector or string scalar.

Example: "CustomVariantParameter_1"

CriterionName — Dependent variant criterion name

character vector | string scalar

Dependent variant criterion name, specified as a character vector or string scalar.

Example: "CustomVariantCriterion_1"

VarAddress — Variant parameter address

string scalar

Starting address for variants of the custom variant parameter, specified as a string scalar. This is an optional parameter.

Example: "0x1234"

Object Functions

<code>add</code>	Add element to ASAP2 file
<code>delete</code>	Remove element from ASAP2 file
<code>find</code>	Filter and get ECU description names
<code>get</code>	Return ASAP2 properties of data element

set Set property for data element

Examples

Create and Add Custom Variant Coding

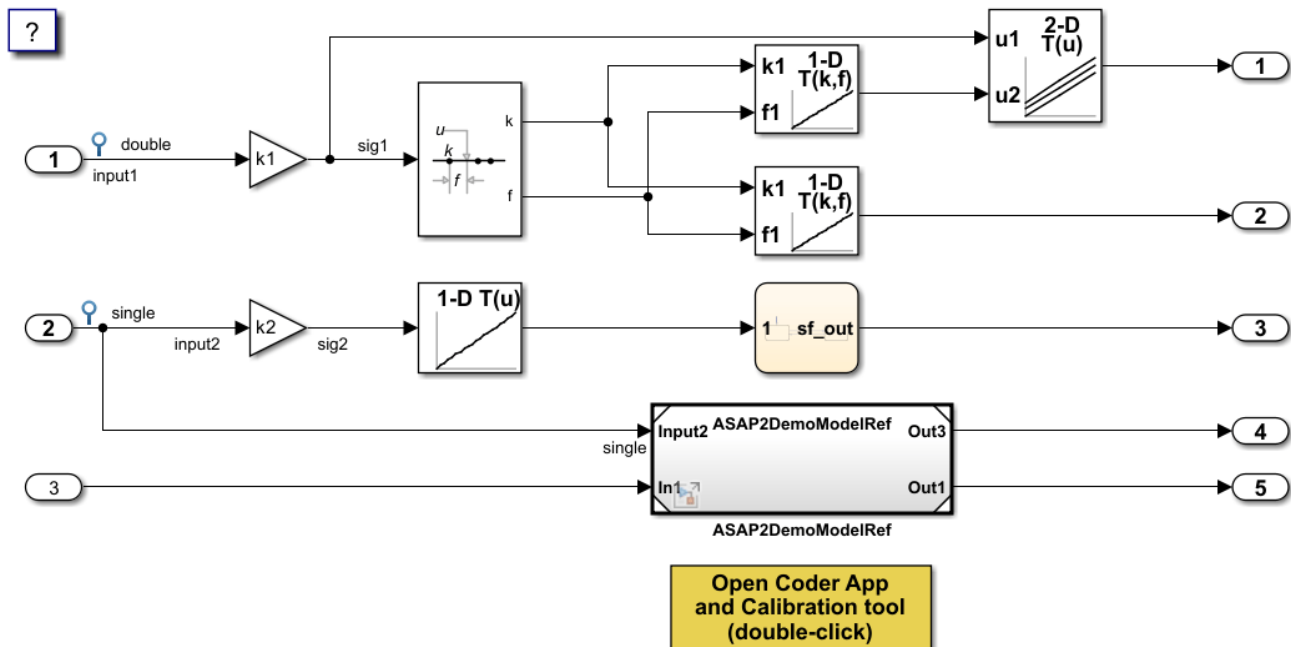
Export a custom variant characteristic object, export a custom variant criterion object, and generate variant coding section in the A2L file.

A model ECU description object has default variant coding section. Exporting a variant characteristic object is mandatory for the variant coding section to appear in the A2L file. You can modify the properties of variant coding section by using the `set` function.

Open and Build Example Model

Open the example model ASAP2Demo.

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");
```

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Verify that there are no variant characteristics or variant criterion available in the description object.

```
find(descObj, "VarCharacteristic")
find(descObj, "VarCriterion")
```

Create and Add Custom Variant Characteristic

To add variant characteristic object to the ASAP2 file, create a custom variant characteristic.

```
VarCharObj = coder.asap2.VarCharacteristic;
VarCharObj.Name = "VehicleSpeedMaxThd";
VarCharObj.CriterionName = ["Fuel", "Transmission"];
VarCharObj.VarAddress = ["0x00087050", "0x00087040", "0x00087068"];
```

Add the custom variant characteristic to the ECU description object.

```
add(descObj, VarCharObj);
```

Add one more custom variant characteristic to the ECU description object.

```
add(descObj, coder.asap2.VarCharacteristic('BatteryVoltageMin'));
```

View Variant Characteristic Properties

Get the properties of the newly added variant characteristic by using this command.

```
get(descObj, 'VarCharacteristic', 'VehicleSpeedMaxThd')
```

Find and Filter Variant Characteristic

Find the variant characteristic objects present in the ECU description object.

```
find(descObj, "VarCharacteristic")
```

Find variant characteristic objects by a property name.

```
find(descObj, "VarCharacteristic", "CriterionName", "Fuel")
```

Update Custom Variant Characteristic Object

Modify the existing variant characteristic VehicleSpeedMaxThd and view the updated properties.

```
set(descObj, "VarCharacteristic", 'VehicleSpeedMaxThd', ...
    'CriterionName', ["Fuel", "Transmission", "Country"], ...
    'VarAddress', ["0x012345", "0x01123AD", "0x1AC4BD", "0x1FFFD"]);
get(descObj, 'VarCharacteristic', 'VehicleSpeedMaxThd')
```

Create and Add Custom Variant Criterion

To add variant criterion object to the ASAP2 file, create a custom variant criterion.

```
VarCriterionObj = coder.asap2.VarCriterion;
VarCriterionObj.Name = 'Fuel';
VarCriterionObj.LongIdentifier = 'Type of Fuel';
VarCriterionObj.Value = ["Electric", "Diesel", "Petrol"];
VarCriterionObj.VarSelectionCharacteristic = 'VarCharSelect_FuelType';
```

Add the custom variant criterion to the ECU description object.

```
add(descObj, VarCriterionObj);
```

Add one more custom variant criterion to the ECU description object.

```
add(descObj,coder.asap2.VarCriterion('Transmission'));
```

View Variant Criterion Properties

Get the properties of the newly added variant criterion by using this command.

```
get(descObj,'VarCriterion','Fuel')
```

Find and Filter Variant Criterion

Find the variant criterion objects present in the ECU description object.

```
find(descObj,'VarCriterion')
```

Find variant criterion objects by a property name.

```
find(descObj,"VarCriterion","VarSelectionCharacteristic","VarCharSelect_FuelType")
```

Update Custom Variant Criterion Object

Modify the existing variant criterion Fuel and view the updated properties.

```
set(descObj,"VarCriterion","Fuel",...  
    'LongIdentifier','Fuel Possibility in Car',...  
    'Value',['Electric',"Petrol"]);  
get(descObj,'VarCriterion','Fuel')
```

Delete Existing Variant Characteristic and Variant Criterion

Remove the newly added variant characteristic and variant criterion from the description object.

```
delete(descObj,'VarCharacteristic','BatteryVoltageMin');  
delete(descObj,'VarCriterion','Transmission');
```

Update Variant Coding Properties

View the properties of default variant coding section.

```
get(descObj,'VariantCoding')
```

Add variant forbidden condition to variant coding object by using the custom data property.

```
sampleString = ["/begin VAR_FORBIDDEN_COMB",...  
    'Fuel Electric',...  
    'Transmission Automatic',...  
    '/end VAR_FORBIDDEN_COMB'];  
set(descObj,'VariantCoding','CustomData',sampleString);
```

Export the ASAP2 file and verify that the variant coding section is present.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions = descObj);
```

Version History

Introduced in R2025a

See Also

`coder.asap2.VarCriterion` | `coder.asap2.getEcuDescriptions` | `add`

Topics

“Generate ASAP2 and CDF Calibration Files”

coder.asap2.VarCriterion

Create variant criterion

Description

Create a custom variant criterion.

Creation

Syntax

```
var_criterion = coder.asap2.VarCriterion
```

Description

`var_criterion = coder.asap2.VarCriterion` creates a variant criterion object. You can use this object to define a custom variant criterion, which can be associated with a variant characteristic.

Properties

Name — Variant criterion name

character vector | string scalar

Variant criterion name, specified as a character vector or string scalar.

Example: "CustomVariantCriterion_1"

LongIdentifier — Description of variant criterion

character vector | string scalar

Description of the variant criterion, specified as a character vector or string scalar.

Example: "Description of variant criterion"

Value — Enumeration of criterion values

character vector | string scalar

Enumeration members of the variant criterion, specified as an array of character vectors or a string array. The enumeration members must be unique within the variant criterion.

Example: ["Car" "Body"]

VarMeasurement — Special measurement object

character vector | string scalar

A special measurement object which indicates the variant running on control software, specified as a character vector or string scalar. This is an optional parameter.

Example: "Measurement_Name"

VarSelectionCharacteristic — Special characteristic object

character vector | string scalar

Specify a special characteristic to change the variant of software running on control unit. This is an optional parameter.

Example: "Characteristic_Name"

Object Functions

add	Add element to ASAP2 file
delete	Remove element from ASAP2 file
find	Filter and get ECU description names
get	Return ASAP2 properties of data element
set	Set property for data element

Examples**Create and Add Custom Variant Coding**

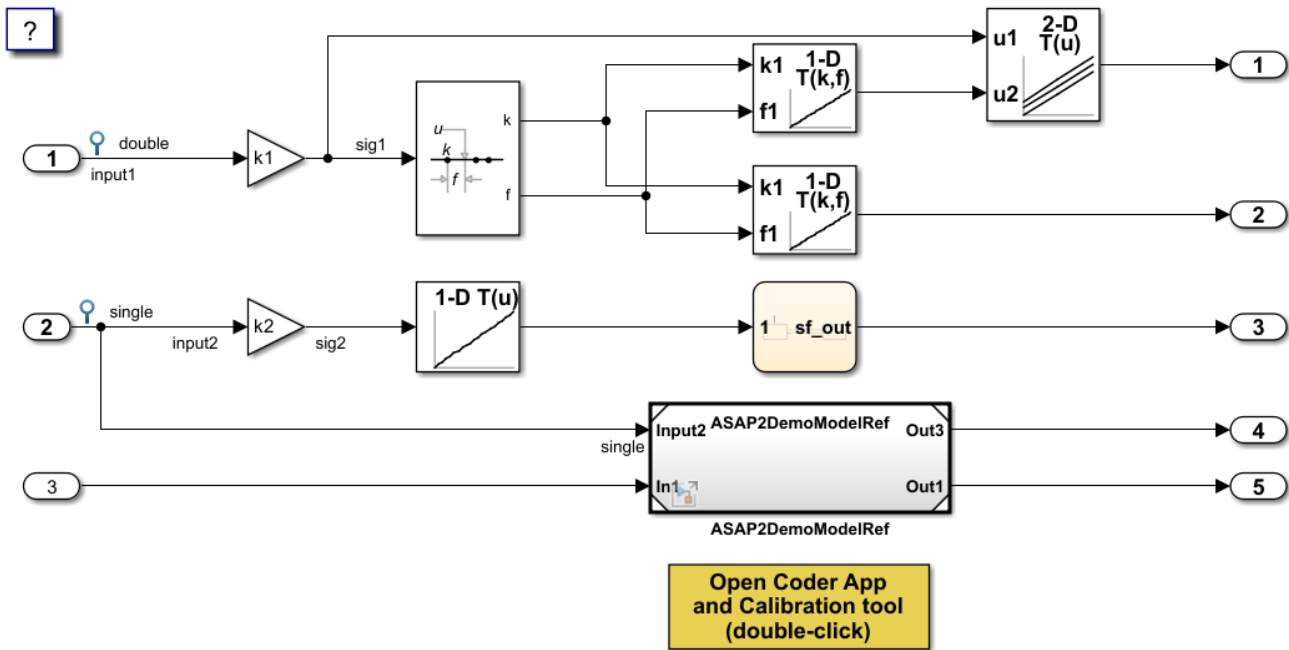
Export a custom variant characteristic object, export a custom variant criterion object, and generate variant coding section in the A2L file.

A model ECU description object has default variant coding section. Exporting a variant characteristic object is mandatory for the variant coding section to appear in the A2L file. You can modify the properties of variant coding section by using the `set` function.

Open and Build Example Model

Open the example model ASAP2Demo.

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");
```

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Verify that there are no variant characteristics or variant criterion available in the description object.

```
find(descObj,"VarCharacteristic")
find(descObj,"VarCriterion")
```

Create and Add Custom Variant Characteristic

To add variant characteristic object to the ASAP2 file, create a custom variant characteristic.

```
VarCharObj = coder.asap2.VarCharacteristic;
VarCharObj.Name = "VehicleSpeedMaxThd";
VarCharObj.CriterionName = ["Fuel","Transmission"];
VarCharObj.VarAddress = ["0x00087050","0x00087040","0x00087068"];
```

Add the custom variant characteristic to the ECU description object.

```
add(descObj,VarCharObj);
```

Add one more custom variant characteristic to the ECU description object.

```
add(descObj,coder.asap2.VarCharacteristic('BatteryVoltageMin'));
```

View Variant Characteristic Properties

Get the properties of the newly added variant characteristic by using this command.

```
get(descObj, 'VarCharacteristic', 'VehicleSpeedMaxThd')
```

Find and Filter Variant Characteristic

Find the variant characteristic objects present in the ECU description object.

```
find(descObj, "VarCharacteristic")
```

Find variant characteristic objects by a property name.

```
find(descObj, "VarCharacteristic", "CriterionName", "Fuel")
```

Update Custom Variant Characteristic Object

Modify the existing variant characteristic VehicleSpeedMaxThd and view the updated properties.

```
set(descObj, "VarCharacteristic", 'VehicleSpeedMaxThd', ...
    'CriterionName', ["Fuel", "Transmission", "Country"], ...
    'VarAddress', ["0x012345", "0x01123AD", "0x1AC4BD", "0x1FFFAD"]);
get(descObj, 'VarCharacteristic', 'VehicleSpeedMaxThd')
```

Create and Add Custom Variant Criterion

To add variant criterion object to the ASAP2 file, create a custom variant criterion.

```
VarCriterionObj = coder.asap2.VarCriterion;
VarCriterionObj.Name = 'Fuel';
VarCriterionObj.LongIdentifier = 'Type of Fuel';
VarCriterionObj.Value = ["Electric", "Diesel", "Petrol"];
VarCriterionObj.VarSelectionCharacteristic = 'VarCharSelect_FuelType';
```

Add the custom variant criterion to the ECU description object.

```
add(descObj, VarCriterionObj);
```

Add one more custom variant criterion to the ECU description object.

```
add(descObj, coder.asap2.VarCriterion('Transmission'));
```

View Variant Criterion Properties

Get the properties of the newly added variant criterion by using this command.

```
get(descObj, 'VarCriterion', 'Fuel')
```

Find and Filter Variant Criterion

Find the variant criterion objects present in the ECU description object.

```
find(descObj, 'VarCriterion')
```

Find variant criterion objects by a property name.

```
find(descObj, "VarCriterion", "VarSelectionCharacteristic", "VarCharSelect_FuelType")
```

Update Custom Variant Criterion Object

Modify the existing variant criterion Fuel and view the updated properties.

```
set(descObj, 'VarCriterion', 'Fuel', ...  
    'LongIdentifier', 'Fuel Possibility in Car', ...  
    'Value', ['Electric', 'Petrol']);  
get(descObj, 'VarCriterion', 'Fuel')
```

Delete Existing Variant Characteristic and Variant Criterion

Remove the newly added variant characteristic and variant criterion from the description object.

```
delete(descObj, 'VarCharacteristic', 'BatteryVoltageMin');  
delete(descObj, 'VarCriterion', 'Transmission');
```

Update Variant Coding Properties

View the properties of default variant coding section.

```
get(descObj, 'VariantCoding')
```

Add variant forbidden condition to variant coding object by using the custom data property.

```
sampleString = ["/begin VAR_FORBIDDEN_COMB", ...  
    'Fuel Electric', ...  
    'Transmission Automatic', ...  
    '/end VAR_FORBIDDEN_COMB'];  
set(descObj, 'VariantCoding', 'CustomData', sampleString);
```

Export the ASAP2 file and verify that the variant coding section is present.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions = descObj);
```

Version History

Introduced in R2025a

See Also

`coder.asap2.VarCharacteristic` | `coder.asap2.getEcuDescriptions` | `add`

Topics

"Generate ASAP2 and CDF Calibration Files"

coder.asap2.getEcuDescriptions

ECU description object for model

Description

An ECU description object and its functions allows you to add, delete, find, filter, update, and fetch ECU descriptions in an ASAP2 file. You can use the function to create the information object for a model.

Creation

`descObj = coder.asap2.getEcuDescriptions(modelName)` creates an ECU description object with name `descObj` for the model `modelName`.

Properties

Comments — Include comments in A2L file

true (default) | false

Generate the A2L file by including or excluding comments.

Example: `Comments=true`

CustomizationObject — Customize ASAP2 fields

`coder.asap2.UserCustomizeBase` object (default)

Create a user base and customize the ASAP2 fields such as:

- `ASAP2FileName`
- `ByteOrder`
- `HeaderComment`
- `ModParComment`
- `ModCommonComment`

Example: `CustomizationObject=obj`

CustomizeGroupsBy — Customize groups in ASAP2 file

ARRAY | ASCII | SCALAR | MAP | CUBOID | CURVE | CUBE_4 | CUBE_5

Group the parameters and signals based on their properties.

Example: `CustomizeGroupsBy={'ARRAY', 'MAP'}`

EcuAddressExtension — Specify additional address information

int

Specify a 32-bit integer value to specify additional address information in the ASAP2 file. By default, no additional address information is required.

Example: `EcuAddressExtension=4`

FileName — Custom name for the exported A2L file

character vector | string scalar

Name for the exported A2L file.

Example: `FileName='test_car'`

Folder — Folder location to export A2L file

character vector | string scalar

Full path to a folder in which to place an exported A2L file.

Example: `Folder='/home/temp/prjct/'`

GenerateXCPIInfo — Include A2ML and IF_DATA in A2L file

true (default) | false

Generate the A2L file by including or excluding A2ML and IF_DATA sections.

Example: `GenerateXCPIInfo=true`

IncludeAllRecordLayouts — Export record layouts of the base data types

false (default) | true

Export record layouts of all the base data types to a file named `RecordLayouts.a2l` according to the version of the A2L file.

Example: `IncludeAllRecordLayouts=true`

IncludeAutosarRteElements — Export RTE elements

true (default) | false

Generate the A2L file by including or excluding the AUTOSAR RTE elements.

Note This option is applicable only for AUTOSAR classic models.

Example: `IncludeAutosarRteElements=true`

IncludeDefaultEventList — Create default event

true (default) | false

Specify true to include a default event list in the A2L file.

Example: `IncludeDefaultEventList=true`

IncludeReferencedModels — Export referenced model elements

true (default) | false

Specify true to include referenced model elements in ASAP2 file. Specify false to exclude.

Example: `IncludeReferencedModels=true`

IncludeSharedElements — Export computation methods and record layouts separately

true (default) | false

Generate separate A2L files for compu methods and record layouts.

Note This option is applicable for models with ERT - based system target files.

Example: IncludeSharedElements=true

IncludeVariantCoding — Export variant parameters

true (default) | false

Specify true to export variant parameters to ASAP2 file. Specify false to skip the variant parameters.

Example: IncludeVariantCoding=true

IndentFile — Follow indentation in A2L file

false (default) | true

Generate an A2L file by following indentation.

Example: IndentFile=true

MapFile — Name of symbol file for model

ELF | PDB | DWARF

Name of the model symbol file that contains symbols of generated code. For example, the addresses of variables used in generated code.

Example: MapFile='model.elf'

ModelClassName — Specify class instance and path names

character vector | string scalar

Custom model instance name in an A2L file. This argument is applicable only for AUTOSAR adaptive models.

Example: ModelClassName='customObj' or
ModelClassName='customNamespace::customObj'

Support64bitIntegers — Include or exclude the 64-bit integers in ASAP2 file

true (default) | false

Specify false to exclude 64-bit integers in the generated ASAP2 file.

Example: Support64bitIntegers=false

SupportStructureElements — Include or exclude the structure elements in ASAP2 file

true (default) | false

Specify false to exclude elements that are part of the structure in the generated ASAP2 file.

Example: SupportStructureElements=false

ToggleArrayLayout — Modify the array layout

false (default) | true

Update the array layout to ROW_DIR or COL_DIR to view the array data in row wise or column wise.

Example: `ToggleArrayLayout=true`

UseSavedSettings — Save and use the preferences

false (default) | true

Save the preferences made and generate the ASAP2 file with the preferences updated such as including the comments, version of the ASAP2 file.

Example: `UseSavedSettings=false`

Version — Version of A2L file

1.71 (default) | 1.31 | 1.61

A2L file format based on the ASAM MCD-2 MC standard defined by ASAM. There are multiple versions of the ASAM MCD-2 MC standard. Specify the version of A2L that you want.

Example: `Version='1.61'` or `Version='1.31'`

Object Functions

add	Add element to ASAP2 file
delete	Remove element from ASAP2 file
find	Filter and get ECU description names
get	Return ASAP2 properties of data element
set	Set property for data element

Examples

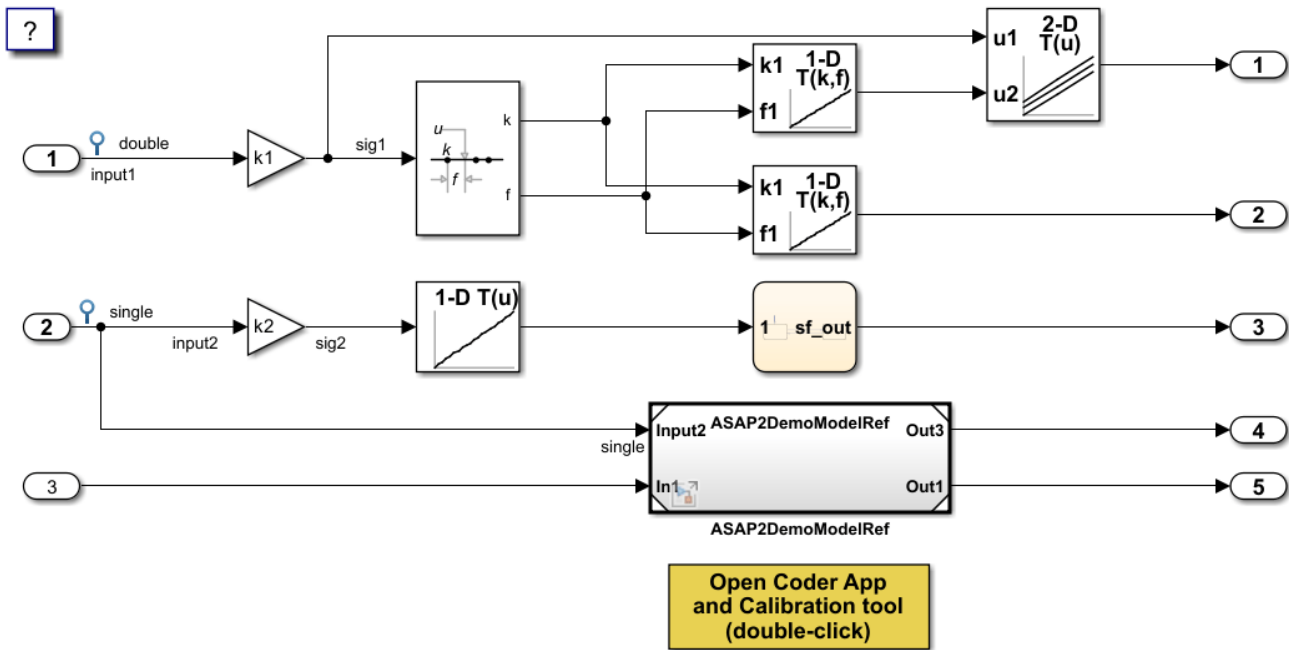
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");
```

```
### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Build Duration
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h 0m 0s

Top model targets:

Model	Build Reason	Status	Build Duration
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h 0m 0s

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

```
Export duration: 0h 0m 0s 165ms
```

Get the list of available computation methods in the description object.

```
find(descObj, "CompuMethod")
```

```
ans = 1x9 string
      "CM_double"      "CM_double_m_per_U0028_s_U005E_2_U0029"      "CM_double_rpm"      "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj, "CompuMethod", Units='rpm')
```

```
ans = 1x3 string
      "CM_double_rpm"      "CM_int16_rpm"      "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj, CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj, "CompuMethod", "CompuMethod_1")
```

```
ans =
  CompuMethod with properties:
      Name: 'CompuMethod_1'
 LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
 CompuVTabValues: [1x1 struct]
    CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj, "CompuMethod", "CompuMethod_1", ConversionType="TAB_VERB");
set(descObj, "CompuMethod", "CompuMethod_1", CompuVTabValues = struct('Literals', ["false" "true"], 'V'
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj, "CompuMethod", "CompuMethod_1")
```

```
modifiedprop =
    CompuMethod with properties:

        Name: 'CompuMethod_1'
        LongIdentifier: 'longIdentifierTest'
        Format: '%2.3'
        Units: 's'
        Coefficients: [2 3]
        ConversionType: "TAB_VERB"
        CompuVTabValues: [1x1 struct]
        CustomData: ""
```

modifiedprop.CompuVTabValues

```
ans = struct with fields:
    Literals: ["false" "true"]
    Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj,"CompuMethod","CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj,Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj,"Characteristic","Custom_parameter1")
```

```
ans =
    Characteristic with properties:

        Name: 'Custom_parameter1'
        LongIdentifier: 'longIdentifierParam'
        Type: 'VALUE'
        EcuAddress: '0x0000'
        CompuMethodName: 'NO_COMPU_METHOD'
        LowerLimit: 0
        UpperLimit: 255
```

```
EcuAddressComment: ""
EcuAddressExtension: []
CalibrationAccess: 'Calibration'
DisplayIdentifier: ""
    Format: ""
    BitMask: []
    AxisInfo: []
RecordLayout: ""
    Dimensions: []
    Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```
ans =
```

```
Measurement with properties:
```

```
    Name: 'Custom_signal1'
LongIdentifier: 'longIdentifierSignal'
   DataType: 'UBYTE'
   EcuAddress: '0x0000'
CompuMethodName: ""
   LowerLimit: 0
   UpperLimit: 255
       Raster: [1x1 struct]
EcuAddressComment: ""
```

```

EcuAddressExtension: []
CalibrationAccess: 'NoCalibration'
DisplayIdentifier: ""
    Format: ""
    BitMask: []
Dimensions: []
    Export: 1
    MaskData: [1x1 struct]
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
CustomData: ""

```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```

LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";

```

Create and add axis information to the parameter.

```

axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;

```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Version History

Introduced in R2022b

See Also

`coder.asap2.export` | `add` | `delete` | `find` | `get` | `set`

Topics

“Generate ASAP2 and CDF Calibration Files”

“Customize Generated ASAP2 File”

“Customize Using Base Files”

“Create and Add Custom Data Elements”

add

Namespace: `coder.asap2`

Add element to ASAP2 file

Syntax

`add(descObj,object)`

Description

`add(descObj,object)` adds the new custom ASAP2 element object to ECU description object `descObj` in order to generate an ASAP2 file with new element.

Examples

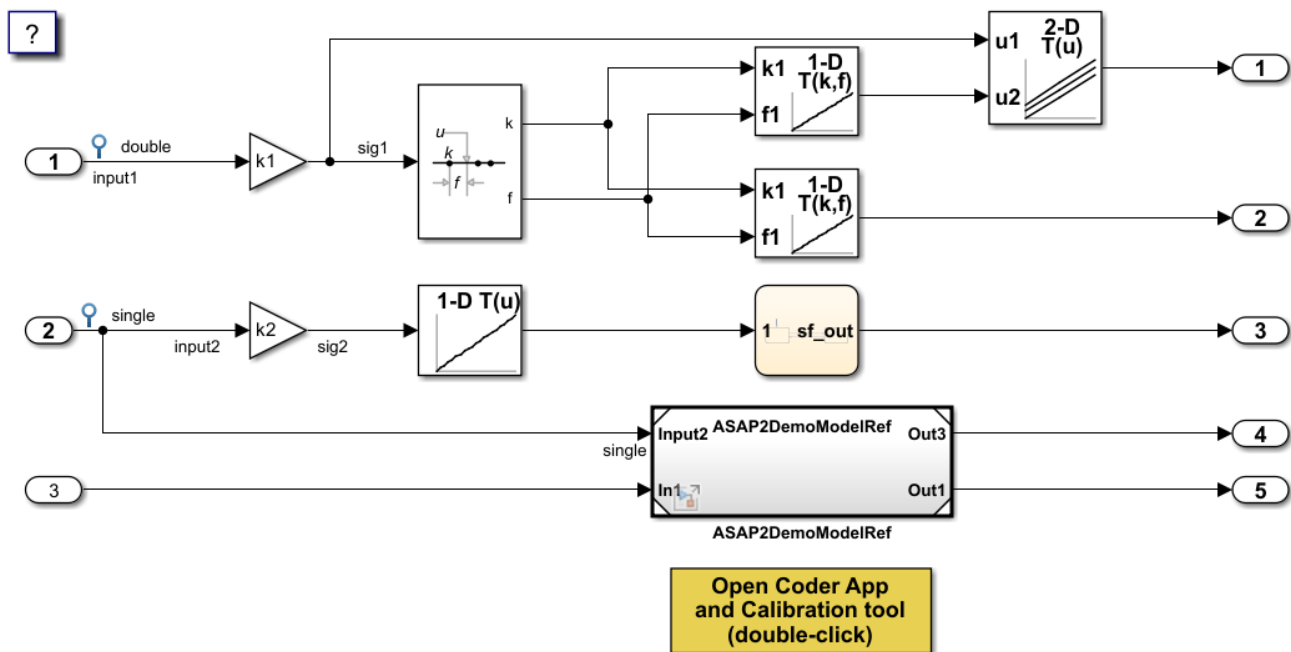
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");

Export duration: 0h 0m 0s 165ms
```

Get the list of available computation methods in the description object.

```
find(descObj,"CompuMethod")

ans = 1x9 string
    "CM_double"    "CM_double_m_per_U0028_s_U005E_2_U0029"    "CM_double_rpm"    "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj,"CompuMethod",Units='rpm')

ans = 1x3 string
    "CM_double_rpm"    "CM_int16_rpm"    "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
```

```
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj, CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj, "CompuMethod", "CompuMethod_1")
```

```
ans =
  CompuMethod with properties:

      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
  CompuVTabValues: [1x1 struct]
      CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj, "CompuMethod", "CompuMethod_1", ConversionType="TAB_VERB");
set(descObj, "CompuMethod", "CompuMethod_1", CompuVTabValues = struct('Literals', ["false" "true"], 'V
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj, "CompuMethod", "CompuMethod_1")
```

```
modifiedprop =
  CompuMethod with properties:

      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: "TAB_VERB"
  CompuVTabValues: [1x1 struct]
      CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:
  Literals: ["false" "true"]
  Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj, "CompuMethod", "CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;  
Parameter_1.Name = 'Custom_parameter1';  
Parameter_1.LongIdentifier = 'longIdentifierParam';  
Parameter_1.UpperLimit = 255;  
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj, Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj, "Characteristic", "Custom_parameter1")
```

```
ans =  
    Characteristic with properties:  
  
        Name: 'Custom_parameter1'  
    LongIdentifier: 'longIdentifierParam'  
        Type: 'VALUE'  
    EcuAddress: '0x0000'  
    CompuMethodName: 'NO_COMPU_METHOD'  
    LowerLimit: 0  
    UpperLimit: 255  
    EcuAddressComment: ''  
    EcuAddressExtension: []  
    CalibrationAccess: 'Calibration'  
    DisplayIdentifier: ''  
        Format: ''  
        BitMask: []  
        AxisInfo: []  
    RecordLayout: ''  
    Dimensions: []  
    Export: 1  
    MaxRefresh: [1x1 struct]  
    SymbolLink: [1x1 struct]  
    CustomData: ''
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```
ans =
    Measurement with properties:
        Name: 'Custom_signal1'
    LongIdentifier: 'longIdentifierSignal'
        DataType: 'UBYTE'
        EcuAddress: '0x0000'
    CompuMethodName: ''
        LowerLimit: 0
        UpperLimit: 255
        Raster: [1x1 struct]
    EcuAddressComment: ''
    EcuAddressExtension: []
    CalibrationAccess: 'NoCalibration'
    DisplayIdentifier: ''
        Format: ''
        BitMask: []
    Dimensions: []
        Export: 1
        MaskData: [1x1 struct]
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ''
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;  
LUT_Parameter.Name = "custom_lookup_table";  
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;  
axis_data(1).Name = 'BP3';  
axis_data(2).Name = 'Bp4';  
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';  
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';  
axis_data(1).MaxAxisPoints = '3';  
axis_data(2).MaxAxisPoints = '3';  
axis_data(1).AxisType = 'STD_AXIS';  
axis_data(2).AxisType = 'STD_AXIS';  
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Input Arguments

descObj — ECU description object

getEcuDescriptions object

Information object created by using the `coder.asap2.getEcuDescriptions` function containing the ASAP2 properties of a given model.

Example: `descObj = coder.asap2.getEcuDescriptions(<modelName>)`

object — Custom ASAP2 element

`coder.asap2.AxisInfo` object | `coder.asap2.Characteristic` object | `coder.asap2.CompuMethod` object | `coder.asap2.Function` object | `coder.asap2.Group` object | `coder.asap2.Measurement` object | `coder.asap2.VarCharacteristic` object | `coder.asap2.VarCriterion` object

Create an object representing ASAP2 element such as characteristic, measurement, axis-point, function, group, or compu method. Add the object to the existing information object and export it to the ASAP2 file.

Example: `CompuMethod_CM1 = coder.asap2.CompuMethod;`
`add(descObj, CompuMethod_CM1);` `coder.asap2.export(<modelName>, CustomEcuDescriptions=descObj)`

Version History

Introduced in R2022b

See Also

`get` | `set` | `find` | `delete` | `coder.asap2.getEcuDescriptions`

delete

Namespace: `coder.asap2`

Remove element from ASAP2 file

Syntax

`delete(descObj,category,name)`

Description

`delete(descObj,category,name)` removes the ASAP2 element of the specified name, category from the `descObj`, and deletes the element from ASAP2 file.

Examples

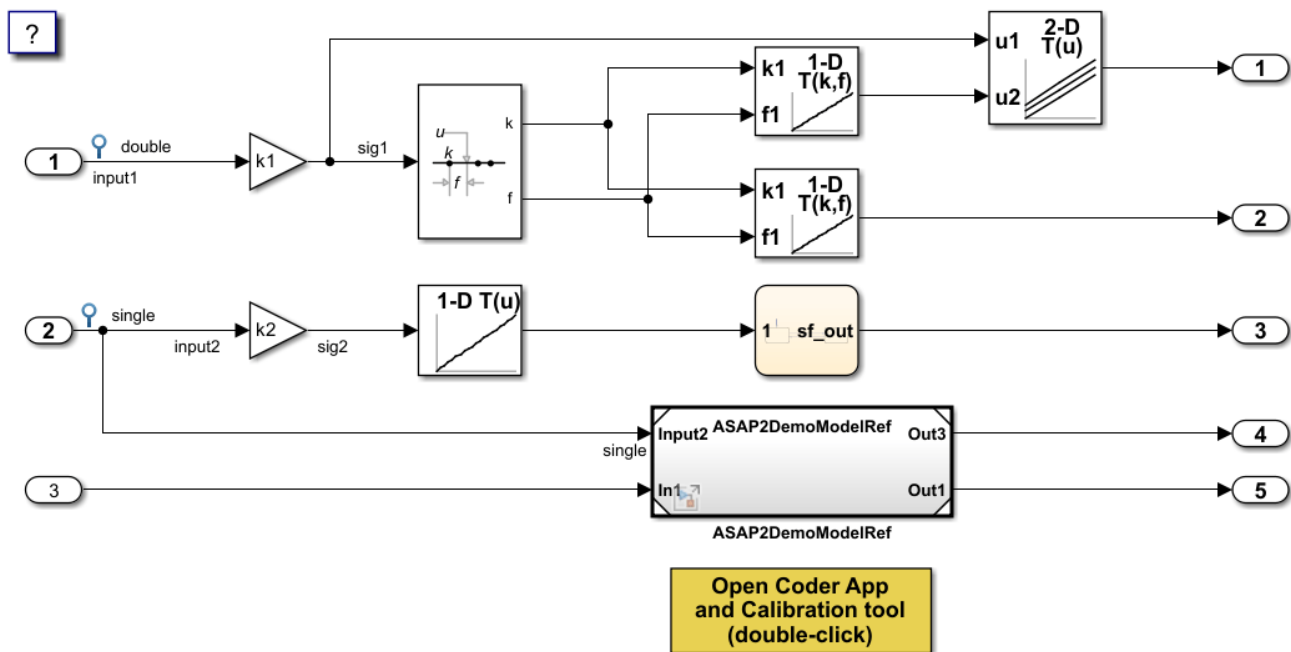
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Export duration: 0h 0m 0s 165ms

Get the list of available computation methods in the description object.

```
find(descObj,"CompuMethod")

ans = 1x9 string
    "CM_double"      "CM_double_m_per_U0028_s_U005E_2_U0029"    "CM_double_rpm"      "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj,"CompuMethod",Units='rpm')

ans = 1x3 string
    "CM_double_rpm"      "CM_int16_rpm"      "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
```

```
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj,CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj,"CompuMethod","CompuMethod_1")
```

```
ans =
    CompuMethod with properties:

        Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
        Format: '%2.3'
        Units: 's'
    Coefficients: [2 3]
    ConversionType: 'LINEAR'
    CompuVTabValues: [1x1 struct]
        CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj,"CompuMethod","CompuMethod_1",ConversionType="TAB_VERB");
set(descObj,"CompuMethod","CompuMethod_1",CompuVTabValues = struct('Literals',["false" "true"],'
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj,"CompuMethod","CompuMethod_1")
```

```
modifiedprop =
    CompuMethod with properties:

        Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
        Format: '%2.3'
        Units: 's'
    Coefficients: [2 3]
    ConversionType: "TAB_VERB"
    CompuVTabValues: [1x1 struct]
        CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:
    Literals: ["false" "true"]
    Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj, "CompuMethod", "CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj, Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj, "Characteristic", "Custom_parameter1")
```

```
ans =
    Characteristic with properties:

        Name: 'Custom_parameter1'
    LongIdentifier: 'longIdentifierParam'
        Type: 'VALUE'
      EcuAddress: '0x0000'
    CompuMethodName: 'NO_COMPU_METHOD'
      LowerLimit: 0
      UpperLimit: 255
    EcuAddressComment: ""
    EcuAddressExtension: []
    CalibrationAccess: 'Calibration'
    DisplayIdentifier: ""
        Format: ""
        BitMask: []
        AxisInfo: []
    RecordLayout: ""
      Dimensions: []
        Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;  
Signal_1.Name = 'Custom_signal1';  
Signal_1.LongIdentifier = 'longIdentifierSignal';  
Signal_1.UpperLimit = 255;  
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```
ans =  
    Measurement with properties:  
  
                Name: 'Custom_signal1'  
    LongIdentifier: 'longIdentifierSignal'  
          DataType: 'UBYTE'  
        EcuAddress: '0x0000'  
    CompuMethodName: ''  
        LowerLimit: 0  
        UpperLimit: 255  
           Raster: [1x1 struct]  
EcuAddressComment: ''  
EcuAddressExtension: []  
  CalibrationAccess: 'NoCalibration'  
    DisplayIdentifier: ''  
           Format: ''  
          BitMask: []  
        Dimensions: []  
           Export: 1  
        MaskData: [1x1 struct]  
      MaxRefresh: [1x1 struct]  
      SymbolLink: [1x1 struct]  
        CustomData: ''
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Input Arguments

descObj — ECU description object

getEcuDescriptions object

Information object created by using the `coder.asap2.getEcuDescriptions` function containing the ASAP2 properties of a given model.

Example: `descObj = coder.asap2.getEcuDescriptions(<modelName>)`

category — Category in ECU descriptions

AxisPoint | Characteristic | CompuMethod | Function | Group | Measurement | VarCharacteristic | VarCriterion

Specify the category of the element to remove it from the ECU description.

Example: `Characteristic`

name — Name from the category

char vector | string scalar

Specify the name available in the specified category to remove it from the ECU description and ASAP2 file.

Example: `Compu_Method1`

Example: `Parameter_p1`

Version History

Introduced in R2022b

See Also

`add` | `get` | `set` | `find` | `coder.asap2.getEcuDescriptions`

find

Namespace: `coder.asap2`

Filter and get ECU description names

Syntax

```
find(descObj,category)
find(descObj,category,property=value)
```

Description

`find(descObj,category)` returns the available ASAP2 element names from the specified category.

`find(descObj,category,property=value)` filters and returns the available ASAP2 element names from the specified category with the specified property set to value.

Examples

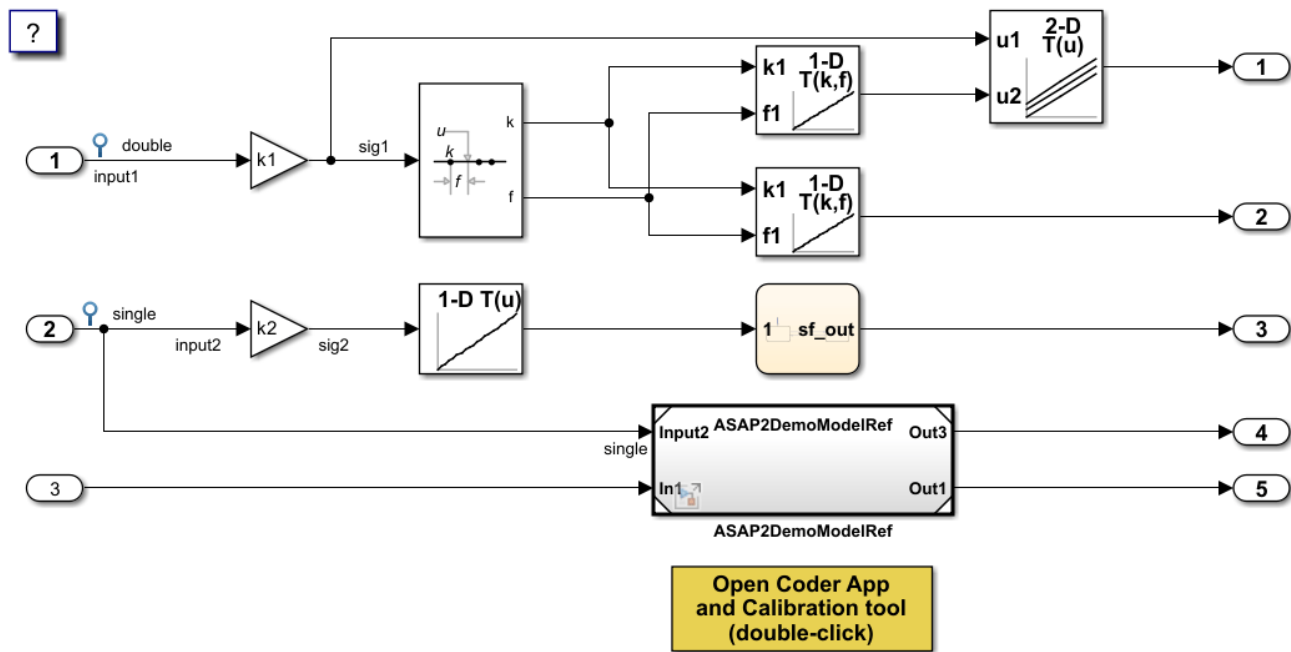
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```


Export duration: 0h 0m 0s 165ms

Get the list of available computation methods in the description object.

```
find(descObj, "CompuMethod")
```

```
ans = 1x9 string
      "CM_double"      "CM_double_m_per_U0028_s_U005E_2_U0029"      "CM_double_rpm"      "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj, "CompuMethod", Units='rpm')
```

```
ans = 1x3 string
      "CM_double_rpm"      "CM_int16_rpm"      "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj, CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj, "CompuMethod", "CompuMethod_1")
```

```
ans =
  CompuMethod with properties:
      Name: 'CompuMethod_1'
 LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
 CompuVTabValues: [1x1 struct]
      CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj, "CompuMethod", "CompuMethod_1", ConversionType="TAB_VERB");
set(descObj, "CompuMethod", "CompuMethod_1", CompuVTabValues = struct('Literals', ["false" "true"], 'V'
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj, "CompuMethod", "CompuMethod_1")
```

```
modifiedprop =  
    CompuMethod with properties:  
  
        Name: 'CompuMethod_1'  
        LongIdentifier: 'longIdentifierTest'  
        Format: '%2.3'  
        Units: 's'  
        Coefficients: [2 3]  
        ConversionType: "TAB_VERB"  
        CompuVTabValues: [1x1 struct]  
        CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:  
    Literals: ["false"      "true"]  
    Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj,"CompuMethod","CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;  
Parameter_1.Name = 'Custom_parameter1';  
Parameter_1.LongIdentifier = 'longIdentifierParam';  
Parameter_1.UpperLimit = 255;  
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj,Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj,"Characteristic","Custom_parameter1")
```

```
ans =  
    Characteristic with properties:  
  
        Name: 'Custom_parameter1'  
        LongIdentifier: 'longIdentifierParam'  
        Type: 'VALUE'  
        EcuAddress: '0x0000'  
        CompuMethodName: 'NO_COMPU_METHOD'  
        LowerLimit: 0  
        UpperLimit: 255
```

```

    EcuAddressComment: ""
    EcuAddressExtension: []
    CalibrationAccess: 'Calibration'
    DisplayIdentifier: ""
        Format: ""
        BitMask: []
        AxisInfo: []
    RecordLayout: ""
        Dimensions: []
        Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""

```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```

Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;

```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```
ans =
```

```
Measurement with properties:
```

```

        Name: 'Custom_signal1'
    LongIdentifier: 'longIdentifierSignal'
        DataType: 'UBYTE'
        EcuAddress: '0x0000'
    CompuMethodName: ""
        LowerLimit: 0
        UpperLimit: 255
        Raster: [1x1 struct]
    EcuAddressComment: ""

```

```
EcuAddressExtension: []
CalibrationAccess: 'NoCalibration'
DisplayIdentifier: ""
    Format: ""
    BitMask: []
Dimensions: []
    Export: 1
    MaskData: [1x1 struct]
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
CustomData: ""
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Input Arguments

descObj — ECU description object

getEcuDescriptions object

Information object created by using the `coder.asap2.getEcuDescriptions` function containing the ASAP2 properties of a given model.

Example: `descObj = coder.asap2.getEcuDescriptions(<modelName>)`

category — Category in ECU descriptions

AxisPoint | Characteristic | CompuMethod | Function | Group | Measurement |
RecordLayout | VarCharacteristic | VarCriterion

Specify the category available in the information object to fetch the available ECU description names.

Example: `Characteristic`

Example: `RecordLayout`

property — Property to filter the ECU description names

char | string

Specify a property and value to filter the list of available ECU description names in the category.

Example: `Units = 'rpm'`

Version History

Introduced in R2022b

See Also

`add` | `get` | `set` | `delete` | `coder.asap2.getEcuDescriptions`

get

Namespace: `coder.asap2`

Return ASAP2 properties of data element

Syntax

`get(descObj, category, name)`

Description

`get(descObj, category, name)` returns the ASAP2 properties of the name for a given category.

Examples

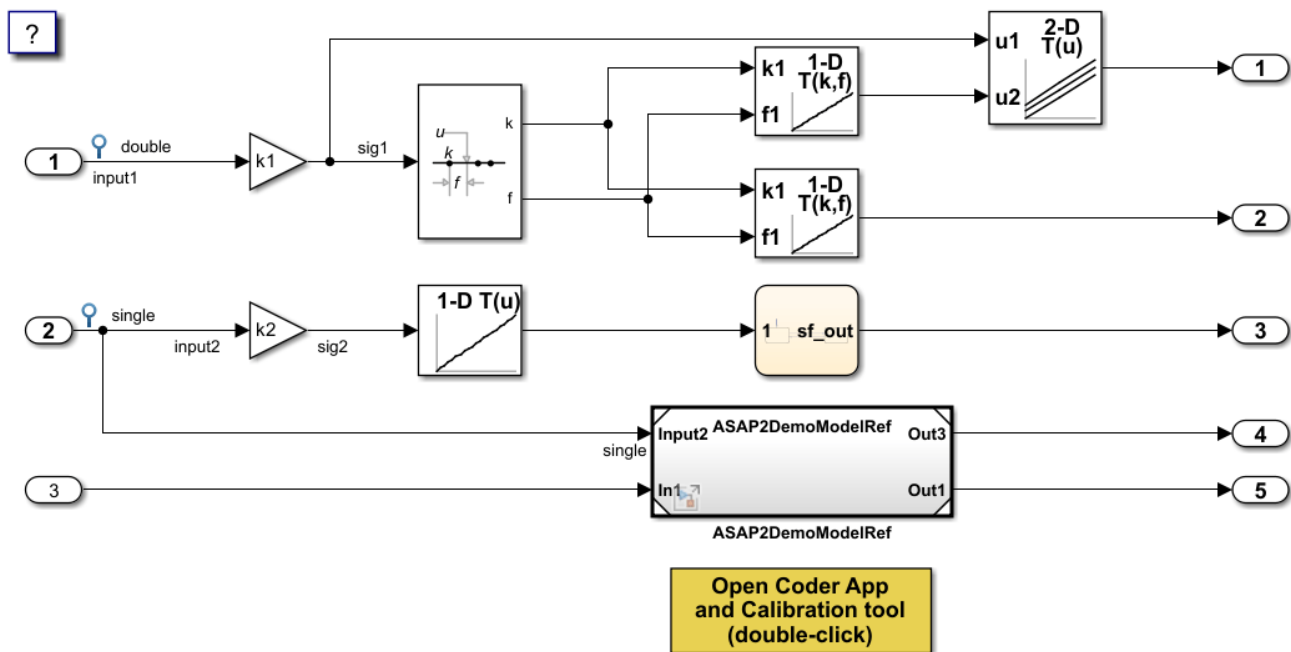
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Export duration: 0h 0m 0s 165ms

Get the list of available computation methods in the description object.

```
find(descObj,"CompuMethod")

ans = 1x9 string
    "CM_double"    "CM_double_m_per_U0028_s_U005E_2_U0029"    "CM_double_rpm"    "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj,"CompuMethod",Units='rpm')

ans = 1x3 string
    "CM_double_rpm"    "CM_int16_rpm"    "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
```

```
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj,CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj,"CompuMethod","CompuMethod_1")
```

```
ans =
  CompuMethod with properties:

      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
  CompuVTabValues: [1x1 struct]
      CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj,"CompuMethod","CompuMethod_1",ConversionType="TAB_VERB");
set(descObj,"CompuMethod","CompuMethod_1",CompuVTabValues = struct('Literals',["false" "true"], 'Values', [0 1]));
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj,"CompuMethod","CompuMethod_1")
```

```
modifiedprop =
  CompuMethod with properties:

      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: "TAB_VERB"
  CompuVTabValues: [1x1 struct]
      CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:
  Literals: ["false" "true"]
  Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```


Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj, "CompuMethod", "CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;
Parameter_1.Name = 'Custom_parameter1';
Parameter_1.LongIdentifier = 'longIdentifierParam';
Parameter_1.UpperLimit = 255;
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj, Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj, "Characteristic", "Custom_parameter1")
```

```
ans =
    Characteristic with properties:

        Name: 'Custom_parameter1'
    LongIdentifier: 'longIdentifierParam'
        Type: 'VALUE'
      EcuAddress: '0x0000'
    CompuMethodName: 'NO_COMPU_METHOD'
      LowerLimit: 0
      UpperLimit: 255
    EcuAddressComment: ""
    EcuAddressExtension: []
    CalibrationAccess: 'Calibration'
    DisplayIdentifier: ""
        Format: ""
        BitMask: []
        AxisInfo: []
    RecordLayout: ""
      Dimensions: []
        Export: 1
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```
ans =
    Measurement with properties:

        Name: 'Custom_signal1'
LongIdentifier: 'longIdentifierSignal'
      DataType: 'UBYTE'
    EcuAddress: '0x0000'
CompuMethodName: ""
    LowerLimit: 0
    UpperLimit: 255
        Raster: [1x1 struct]
EcuAddressComment: ""
EcuAddressExtension: []
CalibrationAccess: 'NoCalibration'
DisplayIdentifier: ""
        Format: ""
        BitMask: []
    Dimensions: []
        Export: 1
    MaskData: [1x1 struct]
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ""
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;
LUT_Parameter.Name = "custom_lookup_table";
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;
axis_data(1).Name = 'BP3';
axis_data(2).Name = 'Bp4';
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';
axis_data(1).MaxAxisPoints = '3';
axis_data(2).MaxAxisPoints = '3';
axis_data(1).AxisType = 'STD_AXIS';
axis_data(2).AxisType = 'STD_AXIS';
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Input Arguments

descObj — ECU description object

getEcuDescriptions object

Information object created by using the `coder.asap2.getEcuDescriptions` function containing the ASAP2 properties of a given model.

Example: `descObj = coder.asap2.getEcuDescriptions(<modelName>)`

category — Category in ASAP2 element properties

AxisPoint | Characteristic | CompuMethod | Function | Group | Measurement | VarCharacteristic | VarCriterion

Specify the category available in the information object to get the ASAP2 element.

Example: `Characteristic`

name — Name from the category

char vector | string scalar

Specify the name available in the specified category to fetch the ASAP2 element.

Example: `CompuMethod1`

Example: `Parameter_p1`

Version History

Introduced in R2022b

See Also

`add` | `set` | `find` | `delete` | `coder.asap2.getEcuDescriptions`

set

Namespace: coder.asap2

Set property for data element

Syntax

```
set(descObj,category,name,Property = value)
```

Description

`set(descObj,category,name,Property = value)` sets the value for the property `Property`, of the name in the category present in ECU description.

Examples

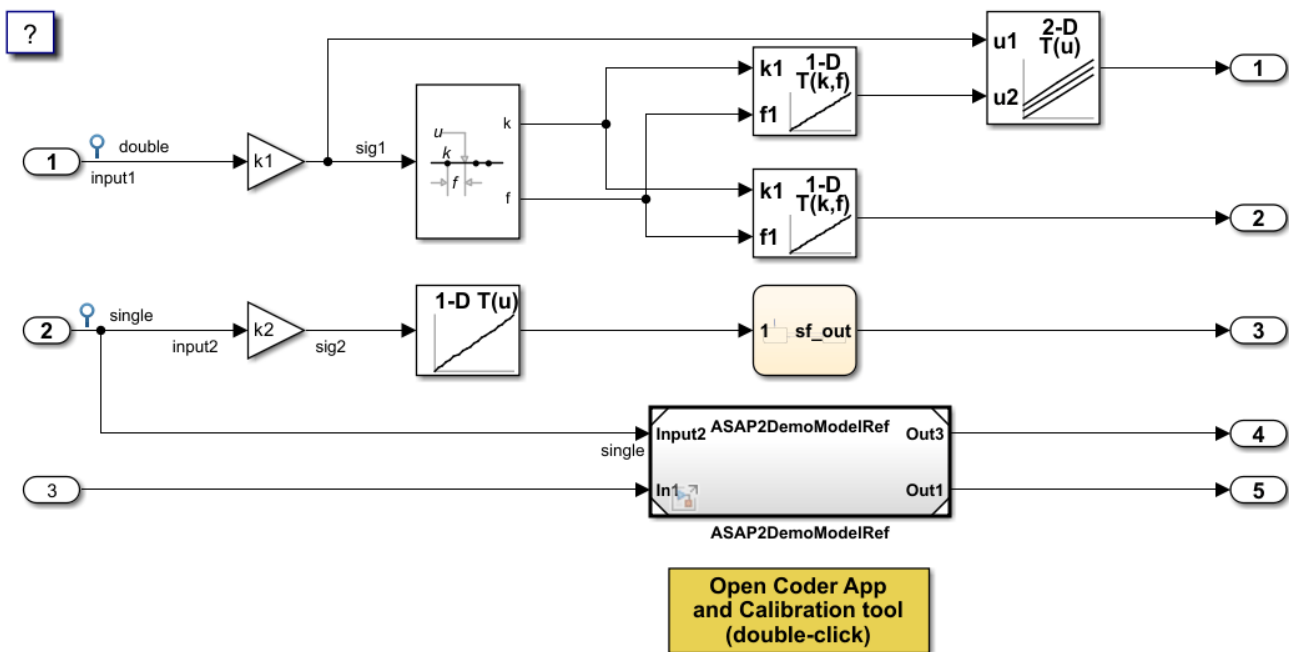
Customize an ASAP2 File

Add, update, filter, find, and remove ECU descriptions in an ASAP2 file.

Open and Build Example Model

Open the example model *ASAP2Demo*

```
open_system("ASAP2Demo");
```



Copyright 1994-2022 The MathWorks, Inc.

Build the model.

```
slbuild("ASAP2Demo");

### Searching for referenced models in model 'ASAP2Demo'.
### Total of 2 models to build.
### Starting serial code generation build.
### Successfully updated the model reference code generation target for: ASAP2DemoModelRef
### Starting build procedure for: ASAP2Demo
### Successful completion of build procedure for: ASAP2Demo
```

Build Summary

Model reference code generation targets:

Model	Build Reason	Status	Bu
ASAP2DemoModelRef	Target (ASAP2DemoModelRef.c) did not exist.	Code generated and compiled.	0h

Top model targets:

Model	Build Reason	Status	Bu
ASAP2Demo	Information cache folder or artifacts were missing.	Code generated and compiled.	0h

2 of 2 models built (0 models already up to date)
Build duration: 0h 0m 23.802s

Create the ECU description object for the model.

```
descObj = coder.asap2.getEcuDescriptions("ASAP2Demo");
```

Export duration: 0h 0m 0s 165ms

Get the list of available computation methods in the description object.

```
find(descObj,"CompuMethod")

ans = 1x9 string
    "CM_double"    "CM_double_m_per_U0028_s_U005E_2_U0029"    "CM_double_rpm"    "CM_int16_rpm"
```

Filter the computation methods and get a list of computation methods that use *rpm* as the unit.

```
find(descObj,"CompuMethod",Units='rpm')

ans = 1x3 string
    "CM_double_rpm"    "CM_int16_rpm"    "CM_single_rpm"
```

Create and Add Custom Computation Method

To add a new computation method to the ASAP2 file, create a custom computation method.

```
CompuMethod_1 = coder.asap2.CompuMethod;
CompuMethod_1.Name = 'CompuMethod_1';
CompuMethod_1.ConversionType = 'LINEAR';
CompuMethod_1.Coefficients = [2 3];
CompuMethod_1.LongIdentifier = 'longIdentifierTest';
```

```
CompuMethod_1.Format = '%2.3';
CompuMethod_1.Units = 's';
```

Add the custom computation method to the ECU description object.

```
add(descObj,CompuMethod_1);
```

Get the properties of the newly added computation method.

```
get(descObj,"CompuMethod","CompuMethod_1")
```

```
ans =
  CompuMethod with properties:
      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: 'LINEAR'
  CompuVTabValues: [1x1 struct]
      CustomData: ""
```

To modify a property of the computation method, use *set* function. Update the *ConversionType* field of the computation method to *TAB_VERB* and define the *CompuVTabValues*.

```
set(descObj,"CompuMethod","CompuMethod_1",ConversionType="TAB_VERB");
set(descObj,"CompuMethod","CompuMethod_1",CompuVTabValues = struct('Literals',["false" "true"],'
```

Get the properties to see the modified fields.

```
modifiedprop = get(descObj,"CompuMethod","CompuMethod_1")
```

```
modifiedprop =
  CompuMethod with properties:
      Name: 'CompuMethod_1'
LongIdentifier: 'longIdentifierTest'
      Format: '%2.3'
      Units: 's'
  Coefficients: [2 3]
  ConversionType: "TAB_VERB"
  CompuVTabValues: [1x1 struct]
      CustomData: ""
```

```
modifiedprop.CompuVTabValues
```

```
ans = struct with fields:
  Literals: ["false" "true"]
  Values: [0 1]
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the computation method *CompuMethod_1*.

```
coder.asap2.export("ASAP2Demo",CustomEcuDescriptions=descObj);
```

Delete Computation Method

Remove the newly added computation method from the description object.

```
delete(descObj, "CompuMethod", "CompuMethod_1");
```

Create and Add Custom Characteristic

To add a new characteristic to the ASAP2 file, create a custom characteristic.

```
Parameter_1 = coder.asap2.Characteristic;  
Parameter_1.Name = 'Custom_parameter1';  
Parameter_1.LongIdentifier = 'longIdentifierParam';  
Parameter_1.UpperLimit = 255;  
Parameter_1.LowerLimit = 0;
```

Add the custom characteristic to the ECU description object.

```
add(descObj, Parameter_1);
```

Get the properties of the newly added characteristic.

```
get(descObj, "Characteristic", "Custom_parameter1")
```

```
ans =  
    Characteristic with properties:  
  
        Name: 'Custom_parameter1'  
    LongIdentifier: 'longIdentifierParam'  
        Type: 'VALUE'  
    EcuAddress: '0x0000'  
    CompuMethodName: 'NO_COMPU_METHOD'  
    LowerLimit: 0  
    UpperLimit: 255  
    EcuAddressComment: ''  
    EcuAddressExtension: []  
    CalibrationAccess: 'Calibration'  
    DisplayIdentifier: ''  
        Format: ''  
        BitMask: []  
        AxisInfo: []  
    RecordLayout: ''  
    Dimensions: []  
    Export: 1  
    MaxRefresh: [1x1 struct]  
    SymbolLink: [1x1 struct]  
    CustomData: ''
```

To modify a property of the characteristic, use *set* function. Update the *UpperLimit* field of the characteristic.

```
set(descObj, "Characteristic", "Custom_parameter1", UpperLimit=128)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the characteristic *Custom_parameter1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```


Remove the newly added characteristic from the description object.

```
delete(descObj, "Characteristic", "Custom_parameter1");
```

Create and Add Custom Measurement

To add a new measurement to the ASAP2 file, create a custom measurement.

```
Signal_1 = coder.asap2.Measurement;
Signal_1.Name = 'Custom_signal1';
Signal_1.LongIdentifier = 'longIdentifierSignal';
Signal_1.UpperLimit = 255;
Signal_1.LowerLimit = 0;
```

Add the custom measurement to the ECU description object.

```
add(descObj, Signal_1);
```

Get the properties of the newly added measurement.

```
get(descObj, "Measurement", "Custom_signal1")
```

```
ans =
    Measurement with properties:

        Name: 'Custom_signal1'
    LongIdentifier: 'longIdentifierSignal'
        DataType: 'UBYTE'
        EcuAddress: '0x0000'
    CompuMethodName: ''
        LowerLimit: 0
        UpperLimit: 255
        Raster: [1x1 struct]
    EcuAddressComment: ''
    EcuAddressExtension: []
    CalibrationAccess: 'NoCalibration'
    DisplayIdentifier: ''
        Format: ''
        BitMask: []
    Dimensions: []
        Export: 1
        MaskData: [1x1 struct]
    MaxRefresh: [1x1 struct]
    SymbolLink: [1x1 struct]
    CustomData: ''
```

To modify a property of the measurement, use *set* function. Update the *CalibrationAccess* field of the measurement.

```
set(descObj, "Measurement", "Custom_signal1", CalibrationAccess='Calibration')
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the measurement with name *Custom_signal1*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Remove the newly added measurement from the description object.

```
delete(descObj, "Measurement", "Custom_signal1");
```

Create and Add Lookup Table Parameter

Add a lookup table parameter.

```
LUT_Parameter = coder.asap2.Characteristic;  
LUT_Parameter.Name = "custom_lookup_table";  
LUT_Parameter.Type = "MAP";
```

Create and add axis information to the parameter.

```
axis_data = coder.asap2.AxisInfo;  
axis_data(1).Name = 'BP3';  
axis_data(2).Name = 'Bp4';  
axis_data(1).CompuMethodName = 'ASAP2Demo_CM_double';  
axis_data(2).CompuMethodName = 'ASAP2Demo_CM_double';  
axis_data(1).MaxAxisPoints = '3';  
axis_data(2).MaxAxisPoints = '3';  
axis_data(1).AxisType = 'STD_AXIS';  
axis_data(2).AxisType = 'STD_AXIS';  
LUT_Parameter.AxisInfo = axis_data;
```

Add the lookup table parameter to the description object.

```
add(descObj, LUT_Parameter)
```

Generate the ASAP2 file using the updated ECU description object and verify that the ASAP2 file contains the lookup table parameter *custom_lookup_table*.

```
coder.asap2.export("ASAP2Demo", CustomEcuDescriptions=descObj);
```

Input Arguments

descObj — ECU description object

getEcuDescriptions object

Information object created by using the `coder.asap2.getEcuDescriptions` function containing the ASAP2 properties of a given model.

Example: `descObj = coder.asap2.getEcuDescriptions(<modelName>)`

category — Category in ECU descriptions

AxisPoint | Characteristic | CompuMethod | Function | Group | Measurement |
VarCharacteristic | VarCriterion

Specify the category available in the information object to set the ASAP2 element.

Example: `Characteristic`

name — Name from the category

char vector | string scalar

Specify the name available in the specified category to set value to a property of the ASAP2 element.

Example: `CompuMethod1`

Example: `Parameter_p1`

Property — Name of the property

char vector | string scalar

Specify a value for the property in the available names from the specified category of the ASAP2 elements.

Example: `descObj.set("CompuMethod","CompuMethod_CM1",Units='custom_units')`

Example: `descObj.set("Characteristic","Parameter_P1",DisplayIdentifier='ParamGain')`

Version History

Introduced in R2022b

See Also

`add` | `get` | `find` | `delete` | `coder.asap2.getEcuDescriptions`

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”.

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.

Version History

Introduced in R2018a

See Also

extmodeEvent | extmodeGetFinalSimulationTime | extmodeInit | extmodeParseArgs |
extmodeReset | extmodeSetFinalSimulationTime | extmodeSimulationComplete |
extmodeStopRequested | extmodeWaitForHostRequest

Topics

“External Mode Simulation by Using XCP Communication”

“Customize XCP Server Software”

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”.

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

`extmodeSimulationTime_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.

Version History

Introduced in R2018a

R2026a: Warns

Warns starting in R2026a

If your target application invokes extmodeEvent using floating-point simulation time values, the external mode simulation issues a warning that support for floating-point simulation time values will be removed in a future release.

See Also

extmodeBackgroundRun | extmodeGetFinalSimulationTime | extmodeInit |
extmodeParseArgs | extmodeReset | extmodeSetFinalSimulationTime |
extmodeSimulationComplete | extmodeStopRequested | extmodeWaitForHostRequest

Topics

“External Mode Simulation by Using XCP Communication”

“Customize XCP Server Software”

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`.

Input Arguments

finalTime — Final simulation time

`extmodeSimulationTime_T` pointer

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.
- `EXTMODE_NOT_INITIALIZED` (-9) -- External mode not initialized yet.

Version History

Introduced in R2018a

R2026a: Warns

Warns starting in R2026a

If your target application invokes `extmodeGetFinalSimulationTime` using floating-point simulation time values, the external mode simulation issues a warning that support for floating-point simulation time values will be removed in a future release.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation by Using XCP Communication”

“Customize XCP Server Software”

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”.

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

extmodeSimulationTime_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.

Version History

Introduced in R2018a

R2026a: Warns

Warns starting in R2026a

If your target application invokes `extmodeInit` using floating-point simulation time values, the external mode simulation issues a warning that support for floating-point simulation time values will be removed in a future release.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` |
`extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` |
`extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation by Using XCP Communication”

“Customize XCP Server Software”

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”.

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”.

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.

Version History

Introduced in R2018a

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation by Using XCP Communication”
“Customize XCP Server Software”

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”.

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.

Version History

Introduced in R2018a

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation by Using XCP Communication”
“Customize XCP Server Software”

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

`extmodeSimulationTime_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.

Version History

Introduced in R2018a

R2026a: Warns

Warns starting in R2026a

If your target application invokes `extmodeSetFinalSimulationTime` using floating-point simulation time values, the external mode simulation issues a warning that support for floating-point simulation time values will be removed in a future release.

See Also

[extmodeBackgroundRun](#) | [extmodeEvent](#) | [extmodeGetFinalSimulationTime](#) | [extmodeInit](#) | [extmodeParseArgs](#) | [extmodeReset](#) | [extmodeSimulationComplete](#) | [extmodeStopRequested](#) | [extmodeWaitForHostRequest](#)

Topics

[“External Mode Simulation by Using XCP Communication”](#)

[“Customize XCP Server Software”](#)

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”.

Output Arguments

simComplete — Simulation complete

true | false

true if model simulation time has reached the specified final simulation time. Otherwise, returns false.

Version History

Introduced in R2018a

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation by Using XCP Communication”
“Customize XCP Server Software”

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”.

Output Arguments

stopRequest — Stop request

true | false

true if request to stop external mode simulation is received. Otherwise, returns false.

Version History

Introduced in R2018a

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation by Using XCP Communication”
“Customize XCP Server Software”

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”.

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.

Version History

Introduced in R2018a

See Also

[extmodeBackgroundRun](#) | [extmodeEvent](#) | [extmodeGetFinalSimulationTime](#) | [extmodeInit](#) | [extmodeParseArgs](#) | [extmodeReset](#) | [extmodeSetFinalSimulationTime](#) | [extmodeSimulationComplete](#) | [extmodeStopRequested](#)

Topics

[“External Mode Simulation by Using XCP Communication”](#)
[“Customize XCP Server Software”](#)

findBuildArg

Namespace: RTW

Find a specific build argument in 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 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 myBuildInfo. Then, view the argument identifier and value.

```
load buildInfo.mat
myBuildInfo = buildInfo;
myBuildArgExtmodeStaticAlloc = 'EXTMODE_STATIC_ALLOC';
[buildArgId buildArgValue] = findBuildArg(buildInfo, ...
    myBuildArgExtmodeStaticAlloc);
```

```
>> buildArgId
```

```
buildArgId =
```

```
    'EXTMODE_STATIC_ALLOC'
```

```
>> buildArgValue
```

```
buildArgValue =
```

```
    '0'
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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 — Build argument name**

character vector | string scalar

Name of the build argument.

value — Build argument value

character vector | string scalar

Value of the build argument.

Version History**Introduced in R2014a****See Also**`getBuildArgs`**Topics**

“Customize Post-Code-Generation Build Processing”

findIncludeFiles

Namespace: RTW

Find and add include (header) files to build information

Syntax

```
findIncludeFiles(buildInfo,extensions,Name,Value)
```

Description

`findIncludeFiles(buildInfo,extensions,Name,Value)` 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 *extensions* 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.

To ensure that `findIncludeFiles` finds header files, add their paths to `buildInfo` by using the `addIncludePaths` function.

Note `findIncludeFiles` can modify the `buildInfo` argument. These modifications are reflected in further use of `buildInfo`, including in subsequent calls to `findIncludeFiles`.

Examples

Find and Add Include Files to Build Information

Find include files with file name extension `.h` that are in the build information, `myBuildInfo`. Add the full paths for these files to the build information. View the include files from the build information.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,{fullfile(pwd,...
    'mycustomheaders')},'myheaders');
findIncludeFiles(myBuildInfo);
headerfiles = getIncludeFiles(myBuildInfo,true,false);
```

```
>> headerfiles
```

```
headerfiles =
```

```
'W:\work\mycustomheaders\myheader.h'
```

Input Arguments

buildInfo — Name of build information object returned by `RTW.BuildInfo` object

Object provides information for compiling and linking generated code.

extensions — 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 *extensions* argument:

- Must start with an asterisk immediately followed by a period (*.)
- Can include a combination of alphanumeric and underscore (_) characters

Example: '*.h' '*.hpp' '*.x*'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `minimalHeaders=false,ignoreParseError=true`

minimalHeaders — Option to include only minimally required header files

true (default) | false

Option to include only minimally required header files, specified as true or false.

Based on the value of `minimalHeaders`, `findIncludeFiles` determines which header files to include and updates `buildInfo` accordingly.

- If `minimalHeaders` is true, `findIncludeFiles` analyzes the source code to determine which header files are required to build the code, and only these header files are included.
- If `minimalHeaders` is false, all header files found on the include paths specified by `buildInfo` are included.

Example: `minimalHeaders=false`

ignoreParseError — Response to parse errors

false (default) | true

If false, terminate on parse errors.

If true, do not terminate on parse errors.

Example: `ignoreParseError=true`

Version History

Introduced in R2006b

See Also

`packNGo` | `addIncludeFiles` | `getIncludeFiles`

Topics

“Customize Post-Code-Generation Build Processing”

getBuildArgs

Namespace: RTW

Get build arguments from build information

Syntax

```
[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs,  
excludeGroupIDs)
```

Description

`[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs,excludeGroupIDs)` returns build argument identifiers and values from 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 project, 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 — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

includeGroupIDs — Group names of build arguments to include in the return from the function

cell array of character vectors | string

Group names of build arguments to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of a build argument, access the `Group` property of the corresponding object.

Example: ''

excludeGroupIDs — Group names of build arguments to exclude from the return from the function

cell array of character vectors | string

Group names of build arguments to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of a build argument, access the `Group` property of the corresponding object.

Example: ''

Output Arguments**identifiers — Build argument names**

cell array of character vectors

Names of the build arguments.

values — Build argument values

cell array of character vectors

Values of the build arguments.

Version History

Introduced in R2014a

See Also

`findBuildArg`

Topics

“Customize Post-Code-Generation Build Processing”

getCodeDescriptor

Create `coder.codescriptor.CodeDescriptor` object for the specified top model

Syntax

```
getCodeDescriptor(model)
getCodeDescriptor(folder)
```

Description

`getCodeDescriptor(model)` creates a `coder.codescriptor.CodeDescriptor` object for the specified model.

`getCodeDescriptor(folder)` creates a `coder.codescriptor.CodeDescriptor` object for the specified build folder.

Examples

Create a Code Descriptor Object Using Model Name

Open the model `CustomCodeComments`.

```
openExample('CustomCodeComments');
```

Build the model.

```
slbuild('CustomCodeComments');
```

Create a `coder.codescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments');
```

Create a Code Descriptor Object Using Build Folder

Open the model `CustomCodeComments`.

```
openExample('CustomCodeComments');
```

Build the model.

```
slbuild('CustomCodeComments');
```

Create a `coder.codescriptor.CodeDescriptor` object by using build folder:

```
codeDescObj = coder.getCodeDescriptor('CustomCodeComments_ert_rtw')
```

Input Arguments

model — Name of the model

string

Model object or name for which to obtain the `coder.codeDescriptor.CodeDescriptor` object.

This method gets the `coder.codeDescriptor.CodeDescriptor` object for the model built as the top model. To get the `coder.codeDescriptor.CodeDescriptor` object for referenced models, use `getReferencedModelCodeDescriptor`. For an example, see “Get Code Descriptor Object for Referenced Model”.

Example: `CustomCodeComments`

Data Types: string

folder — Build folder of the model

string

Build folder of the model for which to obtain the `coder.codeDescriptor.CodeDescriptor` object.

This method gets the `coder.codeDescriptor.CodeDescriptor` object for models built as top models in top model build folders. To get the `coder.codeDescriptor.CodeDescriptor` object for referenced models, use `getReferencedModelCodeDescriptor`.

Example: `C:\Users\Desktop\Work\CustomCodeComments_ert_rtw`

Data Types: string

Version History

Introduced in R2018a

See Also

`coder.codeDescriptor.CodeDescriptor` | `getReferencedModelCodeDescriptor`

Topics

“Get Code Description of Generated Code”

getCompileFlags

Namespace: RTW

Get compiler options from 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 `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-O3'}, ...  
    'OPTS');  
compflags = getCompileFlags(myBuildInfo);
```

```
>> 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 `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-O3'}, ...  
    {'Debug' 'MemOpt'});  
compflags = getCompileFlags(myBuildInfo,'Debug');
```

```
>> compflags
```

```
compflags =
    '-Zi -Wall'
```

Get Compiler Options with Exclude Group Argument

Get the compiler options stored in the build information `myBuildInfo`, except those options with the group name `Debug`.

```
myBuildInfo = RTW.BuildInfo;
addCompileFlags(myBuildInfo,{'-Zi -Wall' '-03'}, ...
    {'Debug' 'MemOpt'});
compflags = getCompileFlags(myBuildInfo, '', 'Debug');
```

```
>> compflags

compflags =

    '-03'
```

Input Arguments

buildinfo — Build information object
object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

includeGroups — Group names of compiler options to include in the return from the function

cell array of character vectors | string

Group names of compiler options to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of a compiler option, access the `Group` property of the corresponding object.

Example: ''

excludeGroups — Group names of compiler options to exclude from the return from the function

cell array of character vectors | string

Group names of compiler options to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of a compiler option, access the `Group` property of the corresponding object.

Example: ''

Output Arguments

options — Compiler options
cell array of character vectors

Compiler options from the build information.

Version History

Introduced in R2006a

See Also

`addCompileFlags` | `getDefines` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing”

getDefines

Namespace: RTW

Get preprocessor macro definitions from 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 myBuildInfo.

```
myBuildInfo = RTW.BuildInfo;
addDefines(myBuildInfo, ...
    {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, 'OPTS');
[defs,names,values] = getDefines(myBuildInfo);
```

```
>> 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 `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addDefines(myBuildInfo, ...  
    {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...  
    {'Debug' 'Debug' 'Debug' 'Release'});  
[defs,names,values] = getDefines(myBuildInfo,'Debug');
```

```
>> defs
```

```
defs =
```

```
    '-DPROTO=first'    '-DDEBUG'    '-Dtest'
```

Get Macro Definitions with Exclude Group Argument

Get the preprocessor macro definitions stored in the build information `myBuildInfo`, except those definitions with the group name `Debug`.

```
myBuildInfo = RTW.BuildInfo;  
addDefines(myBuildInfo, ...  
    {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...  
    {'Debug' 'Debug' 'Debug' 'Release'});  
[defs,names,values] = getDefines(myBuildInfo,'','Debug');
```

```
>> defs
```

```
defs =
```

```
    '-DPRODUCTION'
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

includeGroups — Group names of macro definitions to include in the return from the function

cell array of character vectors | string

Group names of macro definitions to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of a macro definition, access the `Group` property of the corresponding object.

Example: ''

excludeGroups — Group names of macro definitions to exclude from the return from the function

cell array of character vectors | string

Group names of macro definitions to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of a macro definition, access the `Group` property of the corresponding object.

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 ('').

Version History

Introduced in R2006a

See Also

addDefines | getCompileFlags | getLinkFlags

Topics

"Customize Post-Code-Generation Build Processing"

getFullFileList

Namespace: RTW

Get list of files from 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.

To ensure that header files are added to the file list (for example, header files in the `_sharedutils` 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 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 List of Files

After building a project and loading the generated `buildInfo.mat` file, you can list the files stored in the build information object, `buildInfo`. This example returns information for the current project and its submodels.

From the code generation folder that contains the `buildInfo.mat` file, run:

```
bi = load('buildInfo.mat');  
findIncludeFiles(bi.buildInfo);  
[fPathNames, names] = getFullFileList(bi.buildInfo);
```

Get List of Source Files

If you use an `fcase` option, you limit the listing to the files stored in the build information object for the current project. This example returns information for the current project only (not for submodels).

```
[fPathNames,names] = getFullFileList(bi.buildInfo,'source');
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

fcase — File case to return from the build information

' ' (default) | 'source' | 'include' | 'nonbuild'

The optional *fcase* argument specifies whether the function returns the full list for files in the build information or selected cases of files. If you omit the argument or specify a null character vector (' '), the function returns the files from the build information.

Value	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

cell array of character vectors

Fully qualified file paths from the build information.

names — File names

cell array of character vectors

File names from the build information.

Version History

Introduced in R2008a

See Also

findIncludeFiles

Topics

“Customize Post-Code-Generation Build Processing”

getIncludeFiles

Namespace: RTW

Get include files from 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 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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addIncludeFiles(myBuildInfo,{'etc.h' 'etc_private.h' ...  
    'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...  
    '/common/lib'},{'etc' 'etc' 'shared'});  
incfiles=getIncludeFiles(myBuildInfo,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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludeFiles(myBuildInfo,{'etc.h' 'etc_private.h' ...
    'mytypes.h'},{'/etc/proj/etclib' '/etcproj/etc/etc_build' ...
    '/common/lib'},{'etc' 'etc' 'shared'});
incfiles = getIncludeFiles(myBuildInfo,false,false, ...
    'etc');
```

```
>> incfiles
```

```
incfiles =
```

```
    'etc.h'    'etc_private.h'
```

Input Arguments

buildinfo — Build information object

object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

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

Group names of include paths and files to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of an include path or file, access the Group property of the corresponding object.

Example: ''

excl_udeGroups — Group names of include paths and files to exclude from the return from the function

cell array of character vectors | string

Group names of include paths and files to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of an include path or file, access the Group property of the corresponding object.

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 code.

Version History

Introduced in R2006a

See Also

addIncludeFiles | findIncludeFiles | getIncludePaths | getSourceFiles |
getSourcePaths | updateFilePathsAndExtensions

Topics

“Customize Post-Code-Generation Build Processing”

getIncludePaths

Namespace: RTW

Get include paths from 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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,{'/etc/proj/etclib' ...
    '/etcproj/etc/etc_build' '/common/lib'}, ...
    {'etc' 'etc' 'shared'});
incpaths = getIncludePaths(myBuildInfo,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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,{'/etc/proj/etclib' ...
    '/etcproj/etc/etc_build' '/common/lib'}, ...
```

```
{'etc' 'etc' 'shared'}));  
incpaths = getIncludePaths(myBuildInfo,false,'shared');  
  
>> incpaths  
  
incpaths =  
  
    '\common\lib'
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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

Group names of include paths to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of an include path, access the Group property of the corresponding object.

Example: ''

excludeGroups — Group names of include paths to exclude from the return from the function

cell array of character vectors | string

Group names of include paths to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of an include path, access the Group property of the corresponding object.

Example: ''

Output Arguments

paths — Paths of include files

cell array of character vectors

Paths of include files from the build information.

Version History

Introduced in R2006a

See Also

`addIncludePaths` | `getIncludeFiles` | `getSourceFiles` | `getSourcePaths`

Topics

“Customize Post-Code-Generation Build Processing”

getLinkFlags

Namespace: RTW

Get link options from 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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addLinkFlags(myBuildInfo,{'-MD -Gy' '-T'},'OPTS');  
linkflags = getLinkFlags(myBuildInfo);
```

```
>> 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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addLinkFlags(myBuildInfo,{'-MD -Gy' '-T'}, ...  
    {'Debug' 'MemOpt'});  
linkflags = getLinkFlags(myBuildInfo,{'Debug'});
```

```
>> linkflags
```

```
linkflags =
    '-MD -Gy'
```

Get Linker Options with Exclude Group Argument

Get the linker options from the build information myBuildInfo, except those options with the group name Debug.

```
myBuildInfo = RTW.BuildInfo;
addLinkFlags(myBuildInfo,{'-MD -Gy' '-T'}, ...
    {'Debug' 'MemOpt'});
linkflags = getLinkFlags(myBuildInfo, '', {'Debug'});
```

```
>> linkflags

linkflags =
    '-T'
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

includeGroups — Group names of linker options to include in the return from the function

cell array of character vectors | string

Group names of linker options to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of a linker option, access the Group property of the corresponding object.

Example: ''

excludeGroups — Group names of linker options to exclude from the return from the function

cell array of character vectors | string

Group names of linker options to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of a linker option, access the Group property of the corresponding object.

Example: ''

Output Arguments

options — Linker options

cell array of character vectors

Linker options from the build information.

Version History

Introduced in R2006a

See Also

[addLinkFlags](#) | [getCompileFlags](#) | [getDefines](#)

Topics

["Customize Post-Code-Generation Build Processing"](#)

getNonBuildFiles

Namespace: RTW

Get nonbuild-related files from 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 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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myBuildInfo,{'readme.txt' 'myutility1.dll' ...
    'myutility2.dll'});
nonbuildfiles = getNonBuildFiles(myBuildInfo,false,false);
```

```
>> nonbuildfiles
```

```
nonbuildfiles =
```

```
'readme.txt' 'myutility1.dll' 'myutility2.dll'
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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

Group names of non-build files to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of a non-build file, access the Group property of the corresponding object.

Example: ''

excludeGroups — Group names of non-build files to exclude from the return from the function

cell array of character vectors | string

Group names of non-build files to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of a non-build file, access the Group property of the corresponding object.

Example: ''

Output Arguments

files — Non-build files

cell array of character vectors

Names of non-build files from the build information.

Version History

Introduced in R2008a

See Also

`addNonBuildFiles`

Topics

“Customize Post-Code-Generation Build Processing”

getPreincludeFiles

Namespace: RTW

Get preinclude files from build information

Syntax

```
files = getPreincludeFiles(buildinfo, concatenatePaths, replaceMatlabroot,  
includeGroups, excludeGroups)
```

Description

`files = getPreincludeFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)` returns names of preinclude files that are described in the build information.

Examples

Get preinclude Files from Build Information

This example shows how to get preinclude file names from a build information object.

Suppose you have two files, `myFile.c` and `myInc.h`, in your current working folder.

```
// myFile.c  
#include <stdio.h>  
int main(void) {  
    #ifdef __STDC_VERSION__  
        printf("__STDC_VERSION__ is defined\n");  
    #else  
        printf("__STDC_VERSION__ is not defined\n");  
    #endif  
  
    #ifdef MY_MACRO  
        printf("MY_MACRO is defined\n");  
    #else  
        printf("MY_MACRO is not defined\n");  
    #endif  
}
```

```
// myInc.h  
#ifndef MYINC_H  
#define MYINC_H  
  
#define MY_MACRO  
  
#endif
```

Create an `RTW.BuildInfo` object and specify source files, undefine directives, and preinclude files.

```
buildInfo = RTW.BuildInfo;
buildInfo.ComponentName = 'MyComponent';
addSourceFiles(buildInfo, 'myfile.c');
addUndefines(buildInfo, '__STDC_VERSION__');
addPreincludeFiles(buildInfo, 'myinc.h');
```

Get the preinclude filenames.

```
preincFiles=getPreincludeFiles(buildInfo,true,false)
```

```
preincFiles =
```

```
1×1 cell array
```

```
{'myinc.h'}
```

Input Arguments

buildinfo — Build information object

object

Required. RTW.BuildInfo object that contains information for compiling and linking generated code.

concatenatePaths — Concatenate paths and file names

false | true

Required. Specify whether to concatenate paths and filenames that the function returns:

- **true** -- Return list of full paths to preinclude files.
- **false** -- Return only filenames of preinclude files.

replaceMatlabroot — Replace \$(MATLAB_ROOT) and \$(START_DIR) tokens

false | true

Required. Specify whether to replace the \$(MATLAB_ROOT) and \$(START_DIR) tokens:

- **true** -- Replace \$(MATLAB_ROOT) and \$(START_DIR) with the MATLAB installation folder and the code generation folder respectively.
- **false** -- Do not replace \$(MATLAB_ROOT) and \$(START_DIR).

includeGroups — File groups to include

empty (default) | character vector | cell array of character vectors | string scalar

Optional. Specify the file groups that you want to include in the query. To view the group name of a file group, access the Group property of the corresponding object.

excludeGroups — File groups to exclude

empty (default) | character vector | cell array of character vectors | string scalar

Optional. Specify the file groups that you want to exclude from the query. To view the group name of a file group, access the Group property of the corresponding object.

Output Arguments

files — preinclude files

cell array of character vectors

Names of preinclude files that the function obtains from the build information.

Version History

Introduced in R2024a

See Also

[addUndefines](#) | [getUndefines](#) | [removeUndefines](#) | [addPreincludeFiles](#)

Topics

“Define Microsoft Visual Studio Toolchain for Building 32-Bit Application”

getSourceFiles

Namespace: RTW

Get source files from 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 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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo, ...
    {'test1.c' 'test2.c' 'driver.c'},'', ...
    {'Tests' 'Tests' 'Drivers'});
srcfiles = getSourceFiles(myBuildInfo,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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo,{'test1.c' 'test2.c'...
    'driver.c'}, {'/proj/test1' '/proj/test2'...
    '/drivers/src'}, {'tests', 'tests', 'drivers'});
incfiles = getSourceFiles(myBuildInfo,false,false,...
    'tests');
```

```
>> incfiles
```

```
incfiles =
```

```
    'test1.c'    'test2.c'
```

Input Arguments

buildinfo — Build information object

object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

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 <code>\$(MATLAB_ROOT)</code> with the absolute path for your MATLAB installation folder.
false	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

Example: true

includeGroups — Group names of source files to include in the return from the function

cell array of character vectors | string

Group names of source files to include in the return from the function, specified as a string or a cell array of character vectors. To view the group name of a source file, access the `Group` property of the corresponding object.

Example: ''

excludeGroups — Group names of source files to exclude from the return from the function
cell array of character vectors | string

Group names of source files to exclude from the return from the function, specified as a string or a cell array of character vectors. To view the group name of a source file, access the Group property of the corresponding object.

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 addIncludeFiles function. If you call the packNGo function, the names include files that packNGo found and added while packaging code.

Version History

Introduced in R2006a

See Also

addSourceFiles | getIncludeFiles | getIncludePaths | getSourcePaths |
updateFilePathsAndExtensions

Topics

“Customize Post-Code-Generation Build Processing”

getSourcePaths

Namespace: RTW

Get source paths from 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 returns only the file paths that were added to the build information by using `addSourcePaths`. The build process uses build information source paths to locate source files that were specified without an explicit path.

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, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;  
addSourcePaths(myBuildInfo,{'/proj/test1' ...  
    '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...  
    'drivers'});  
srcpaths = getSourcePaths(myBuildInfo,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, `myBuildInfo`.


```

myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,{'/proj/test1' ...
    '/proj/test2' '/drivers/src'}, {'tests' 'tests' ...
    'drivers'});
srcpaths = getSourcePaths(myBuildInfo,true,'tests');

>> srcpaths

srcpaths =

    '\proj\test1'    '\proj\test2'

```

Get Source Paths from Build Information

Get a source path from the build information, `myBuildInfo`. 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`.

```

myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo, fullfile(matlabroot, ...
    'rtw', 'c', 'src'));
srcpaths = getSourcePaths(myBuildInfo,false);

>> srcpaths{:}

ans =

$(MATLAB_ROOT)\rtw\c\src

>> srcpaths = getSourcePaths(myBuildInfo,true);
>> srcpaths{:}

ans =

\\myserver\myworkspace\matlab\rtw\c\src

```

Input Arguments

buildinfo — Name of the build information object returned by `RTW.BuildInfo`
object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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

Group names of source paths to include in the return from the function, specified as a string or cell array of character vectors. To view the group name of a source path, access the Group property of the corresponding object.

Example: ''

excludeGroups — Group names of source paths to exclude from the return from the function

cell array of character vectors | string

Group names of source paths to exclude from the return from the function, specified as a string or cell array of character vectors. To view the group name of a source path, access the Group property of the corresponding object.

Example: ''

Output Arguments

srcpaths — Source file paths

cell array of character vectors

Paths of source files from the build information.

Version History

Introduced in R2006a

See Also

addSourcePaths | getIncludeFiles | getIncludePaths | getSourceFiles

Topics

“Customize Post-Code-Generation Build Processing”

getUndefines

Namespace: RTW

From build information, get preprocessor macros to undefine

Syntax

```
undefines = getUndefines(buildinfo,includeGroups,excludeGroups)
```

Description

`undefines = getUndefines(buildinfo,includeGroups,excludeGroups)` gets, from the build information, preprocessor macros to undefine.

Examples

Get undefine Preprocessor Macros from Build Information

This example shows how to get, from a build information object, preprocessor macros to undefine.

Suppose you have two files, `myFile.c` and `myInc.h`, in your current working folder.

```
// myFile.c
#include <stdio.h>
int main(void) {
    #ifdef __STDC_VERSION__
        printf("__STDC_VERSION__ is defined\n");
    #else
        printf("__STDC_VERSION__ is not defined\n");
    #endif

    #ifdef MY_MACRO
        printf("MY_MACRO is defined\n");
    #else
        printf("MY_MACRO is not defined\n");
    #endif
}
```

```
// myInc.h
#ifndef MYINC_H
#define MYINC_H

#define MY_MACRO

#endif
```

Create an `RTW.BuildInfo` object and specify source files, undefine directives, and preinclude files.

```
buildInfo = RTW.BuildInfo;
buildInfo.ComponentName = 'MyComponent';
```

```
addSourceFiles(buildInfo, 'myfile.c');  
addUndefines(buildInfo, '__STDC_VERSION__');  
addPreincludeFiles(buildInfo, 'myinc.h');
```

Get the preprocessor macros to undefine.

```
undefines=getUndefines(buildInfo)
```

```
undefines =
```

```
    1×1 cell array
```

```
    {'__STDC_VERSION__'}
```

Input Arguments

buildinfo — Build information object

object

Required. RTW.BuildInfo object that contains information for compiling and linking generated code.

includeGroups — Macro groups to include

empty (default) | character vector | cell array of character vectors | string scalar

Optional. Specify the undefine preprocessor macro groups that you want to include in the query. To view the group name of a macro group, access the Group property of the corresponding object.

excludeGroups — Macro groups to exclude

empty (default) | character vector | cell array of character vectors | string scalar

Optional. Specify the undefine preprocessor macro groups that you want to exclude from the query. To view the group name of a macro group, access the Group property of the corresponding object.

Output Arguments

undefines — Preprocessor macros

cell array of character vectors

Preprocessor macros to undefine, which the function obtains from the build information.

Version History

Introduced in R2024a

See Also

`addUndefines` | `removeUndefines` | `addPreincludeFiles` | `getPreincludeFiles`

Topics

“Define Microsoft Visual Studio Toolchain for Building 32-Bit Application”

model_initialize

Generated C/C++ entry-point function that contains initialization code for a Simulink model

Syntax

```
void model_initialize(void)
```

Description

`void model_initialize(void)` is a generated C or C++ entry-point function called one time to execute the initialization code for a Simulink model. This function is not intended to reset the real-time model data structure (rtM) for a model.

The generated calling interface of the initialize entry-point function for a model differs depending on the **Language** and **Code interface packaging** parameters. For more information, see [Code interface packaging \(component\)](#).

To preview and customize the name of a generated C initialize entry-point function an Embedded Coder license is required. To preview the function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function using a template, in the **Function Customization Template** column select a template to apply to the function. For more information, see “Customize Generated Entry-Point C Function Names” and “Configure Default Code Generation for Functions”.

To view the generated initialize entry-point function, open the **Code** view or Code Generation Report and examine the source code for your model. For more information see, “Analyze Generated Data Code Interface Report”.

Note Do not confuse `model_initialize` with other initialization functions that might be generated for referenced models, such as `model_Init`. Those initialization functions do not appear in the Code Mappings editor and are not customizable.

Examples

Generate An Initialize Entry-Point Function

This example shows the basic workflow for how to configure, customize, generate, and examine an initialize entry-point function.

- 1 Open a model.
- 2 Select a coder. In the Apps gallery, click **Simulink Coder** or **Embedded Coder**.
- 3 Set model configuration parameters **Language** and **Code interface packaging**.
- 4 (Embedded Coder only) Customize the function. Using Embedded Coder, you can customize the name of the initialize entry-point function.
 - Open the Code Mappings editor.

- Click the **Functions** tab.
 - In the **Function Name** column, enter a name for the function.
- 5 Generate code.
 - 6 Examine the generated code. In the **Code** view, verify that the generated initialize function appears with the expected name and parameters.

Input Arguments

void — Default initialize function parameter

void (default)

The initialize entry-point C or C++ function provides an interface to start application code. By default, the generated function provides a **void-void** interface that does not have arguments.

Example: void

Output Arguments

void — Initialize function return value

void (default)

The initialize entry-point C or C++ function provides an interface to start application code. By default, the generated function provides a **void-void** interface that does not have a return value.

Example: void

Version History

Introduced before R2006a

See Also

`model_step` | `model_terminate` | `model_reset`

Topics

“Customize Generated Entry-Point C Function Names”

“Configure Generated C Function Interface for Model Entry-Point Functions”

“Configure Names for Individual C Entry-Point Functions”

“Configure Default Code Generation for Functions”

“Analyze Generated Data Code Interface Report”

“Startup, Reset, and Shutdown Function Interfaces”

model_Init

Generated C/C++ function that contains part of the initialization code for referenced Simulink models

Syntax

```
void model_Init(void)
```

Description

`void model_Init(void)` is a C or C++ function generated for referenced Simulink models. `model_Init` is invoked by other generated functions as part of the initialization of the model, such as the initialization of state conditions. It does not appear in the Code Mappings editor and cannot be customized. Do not confuse `model_Init` with `model_initialize`.

Version History

Introduced before R2006a

model_reset

Generated C/C++ entry-point function that contains reset code for a Simulink model

Syntax

```
void model_reset(void)
```

Description

`void model_reset(void)` is a generated C or C++ entry-point function that executes reset code for a Simulink model. If a model includes a Reset Function block, reset code is generated. To reset conditions or state, call the function from the application code.

The generated calling interface of the reset entry-point function for a model differs depending on the **Language** and **Code interface packaging** parameters. For more information, see [Code interface packaging \(component\)](#).

To preview and customize the name of the generated reset entry-point function an Embedded Coder license is required. To preview the reset entry-point function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function name and arguments, in the **Function Preview** column click the function hyperlink and configure the reset function from the opened dialogue box. To customize a function using a template (not supported for C++), in the **Function Customization Template** column select a template to apply to the function. For more information, see ["Customize Generated Entry-Point C Function Names"](#), ["Interactively Configure C++ Interface"](#), and ["Configure Default Code Generation for Functions"](#).

To view the generated reset entry-point function, open the **Code** view or Code Generation Report and view the source code for your model. For more information see, ["Analyze Generated Data Code Interface Report"](#).

Examples

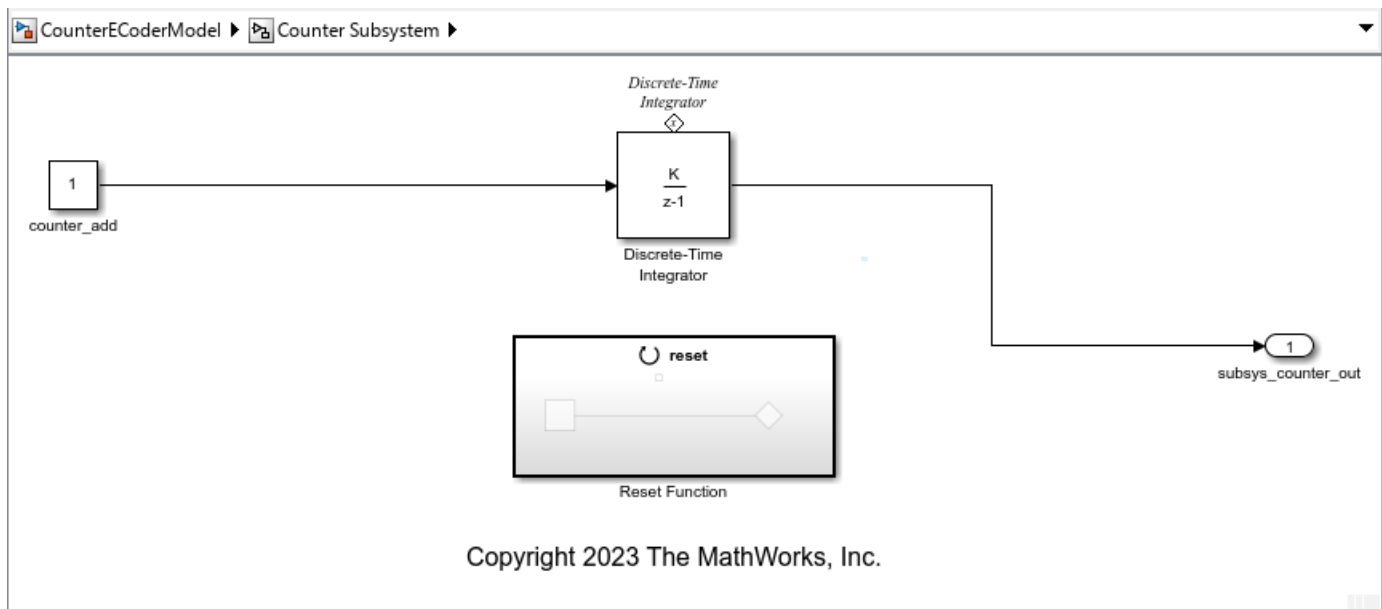
Generate Reset Entry-Point Function with Embedded Coder®

This example shows how to generate a nonreusable C reset entry-point function with Embedded Coder.

Examine Counter Model and Reset Function

Open the counter model and examine the subsystem Counter Subsystem. The counter is driven by a Discrete-Time Integrator block.

```
counterModel = "CounterECoderModel";  
open_system(counterModel);
```

Simulate the model and observe how the subsystem increments the output. The simulation is set to run in paced mode to slow down the counting in this example.

```
sim(counterModel);
```

Generate C code from the model.

```
evalc("slbuild(counterModel,GenerateCodeOnly=true);");
```

Open the generated .c file and inspect the generated reset function CounterECoderModel_reset.

```
generatedCFile = fullfile(counterModel + "_ert_rtw",counterModel + ".c")
```

```
generatedCFile =  
"CounterECoderModel_ert_rtw\CounterECoderModel.c"
```

The reset function sets the value of the Discrete-Time Integrator block to 0.

```
coder.example.extractLines(generatedCFile,"CounterE_CounterSubsystem_reset(void)",...  
    "/* End of Outputs for SubSystem: '<S1>/Reset Function' */",true,true);
```

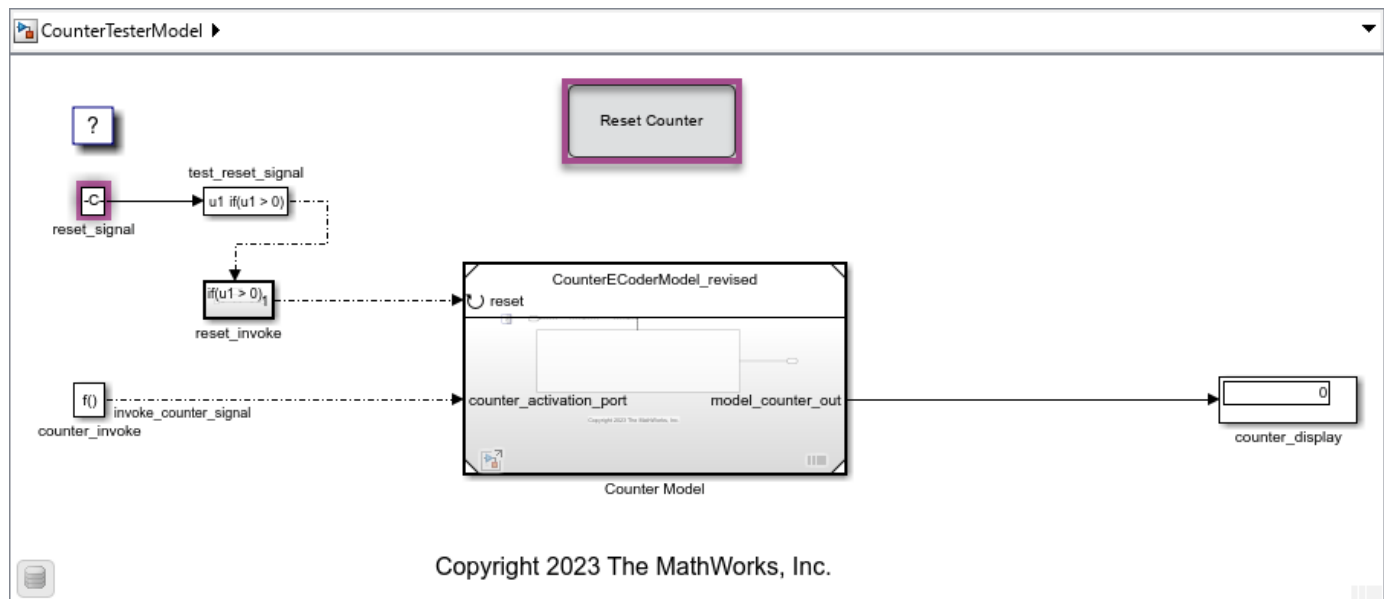
```
void CounterE_CounterSubsystem_reset(void)  
{  
    /* Outputs for Atomic SubSystem: '<S1>/Reset Function' */  
    /* StateWriter: '<S2>/State Writer' incorporates:  
     * Constant: '<S2>/Constant'  
     */  
    CounterECoderModel_DW.DiscreteTimeIntegrator_DSTATE = 0.0;  
  
    /* End of Outputs for SubSystem: '<S1>/Reset Function' */
```

Use Tester Model to Test Counter Model and Reset Function

In this part of the example a tester model acts as a wrapper that references and feeds a revised version of the counter model. The tester model can invoke the functions of the counter model during simulation.

Open the tester model. The tester model feeds the counter activation trigger of the counter model, which runs together with the tester model throughout the simulation. The Reset Counter button of the tester model feeds the reset function of the counter model. Clicking this button resets the counter.

```
testerModel = "CounterTesterModel";
open_system(testerModel);
```



Simulate the tester model and observe the counting that it captures from the counter model.

```
sim(testerModel);
```

Click the Reset Counter button to invoke the reset entry point function. The counter is reset. You can click the Reset Counter button multiple times.

Input Arguments

void — Default reset function parameter

void (default)

The reset entry-point C or C++ function provides an interface to the model reset code. By default, the generated function provides a void-void interface that does not have arguments.

Output Arguments

void — Reset function return value

void

The reset entry-point C or C++ function provides an interface to model reset code. By default, the generated function provides a void-void interface that does not have a return value.

Version History

Introduced before R2006a

See Also

`model_initialize` | `model_step` | `model_terminate`

Topics

"Customize Generated Entry-Point C Function Names"

"Configure Generated C Function Interface for Model Entry-Point Functions"

"Configure Name and Arguments for Individual Step Functions"

"Interactively Configure C++ Interface"

"Configure Default Code Generation for Functions"

"Analyze Generated Data Code Interface Report"

"Startup, Reset, and Shutdown Function Interfaces"

model_step

Generated C/C++ entry-point function that contains execution code for each step in a Simulink model

Syntax

```
void model_step(void)
void model_step_N(void)
```

Description

`void model_step(void)` is an execution function that contains the output and update code for the blocks in a Simulink model.

`void model_step_N(void)` is an execution function with a task identifier that contains the output and update code for the blocks in a Simulink model.

The step entry-point function computes the current values of the blocks. If logging is enabled, the step function updates logging variables. By design, the step function is called at the interrupt level from `rt_OneStep` (invoked as a timer ISR). The `rt_OneStep` function calls the `model_step` function to execute processing for one clock period of the model. For a more information, see “`rt_OneStep` and Scheduling Considerations”.

If the model has a finite stop time, the step function signals the end of execution when the current time equals the stop time. Otherwise, if one or more of these conditions are true, the step function does not check the current time against the stop time and the program runs indefinitely:

- The model stop time is set to `inf`.
- Logging is disabled.
- Parameter **Terminate function required** is not selected.

The generated calling interface of the step entry-point function for a model differs depending on these parameters:

- 1 To generate a step entry-point function, select the **Single output/update function** parameter. If you clear this parameter, `model_output` and `model_update` entry-point functions are generated in place of the step function.
- 2 To generate a single step function with configurable arguments, clear the **Treat each discrete rate as a separate task** parameter. To generate separate step functions based on timing requirements, select this parameter. For more information, see `Treat each discrete rate as a separate task`.
 - Parameter Value: `Off` (single rate or multirate single-tasking model)
Function Prototype: `void model_step(void);`
 - Parameter Value: `On` (multirate multi-tasking model)
Function Prototype: `void model_step_N (void);` (N is a task identifier)
- 3 To change the generated calling interface, set the **Language** and **Code interface packaging** parameters. For more information, see `Code interface packaging` (component).

To preview and customize the name and arguments of the generated C or C++ step entry-point function an Embedded Coder license is required. To preview a step entry-point function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function name and arguments, in the **Function Preview** column click the function hyperlink and configure the step function from the opened dialog box. To customize a function using a template, in the **Function Customization Template** column select a template to apply to the function. For more information, see “Customize Generated Entry-Point C Function Names”, “Interactively Configure C++ Interface”, and “Configure Default Code Generation for Functions”.

To view the generated step entry-point function, open the **Code** view or Code Generation Report and view the source code for your model. For more information see, “Analyze Generated Data Code Interface Report”.

Examples

Generate A Step Entry-Point Function

This example shows the basic workflow for how to configure, customize, generate, and examine a step entry-point function.

- 1 Open a model.
- 2 Select a coder. In the Apps gallery, click **Simulink Coder** or **Embedded Coder**.
- 3 Set model configuration parameters **Single output/update function**, **Treat each discrete rate as a separate task**, **Language**, and **Code interface packaging**.
- 4 (Embedded Coder only) Customize the function. Using Embedded Coder, you can customize the name and arguments of the step entry-point function.
 - Open the Code Mappings editor.
 - Click on the **Functions** tab.
 - Customize the name and arguments. In the **Function Preview** column, click the function hyperlink to open the **Configure C Step Function Interface** dialog box. Configure the name and arguments.
- 5 Generate code.
- 6 Examine the generated code. In the **Code** view, verify the generated step function appears with the expected name and parameters.

Input Arguments

void — Default step function parameter

void (default)

The step entry-point C or C++ function provides an interface to the model execution code. By default, the generated function provides a **void-void** interface that does not have arguments. To configure the input arguments for a C step function, use the Code Mappings Editor - C. To configure the input arguments for a C++ step function, use the Code Mappings - C++ Editor.

Output Arguments

void — Step function return value

void

The step entry-point C or C++ function provides an interface to the model execution code. By default, the generated function provides a `void-void` interface that does not have arguments. To configure the output arguments for a C step function, use the Code Mappings Editor - C. To configure the output arguments for a C++ step function, use the Code Mappings - C++ Editor.

Version History

Introduced before R2006a

See Also

`model_initialize` | `model_terminate` | `model_reset`

Topics

“Customize Generated Entry-Point C Function Names”

“Configure Generated C Function Interface for Model Entry-Point Functions”

“Configure Name and Arguments for Individual Step Functions”

“Interactively Configure C++ Interface”

“Configure Default Code Generation for Functions”

“Analyze Generated Data Code Interface Report”

“Startup, Reset, and Shutdown Function Interfaces”

model_terminate

Generated C/C++ entry-point function that contains termination code for a Simulink model

Syntax

```
void model_terminate(void)
```

Description

`void model_terminate(void)` is a generated C or C++ entry-point function called one time to execute termination code for a Simulink model.

The generated calling interface of the termination entry-point function for a model differs depending on the **Language** and **Code interface packaging** parameters. For more information, see Code interface packaging (component). With Embedded Coder, you can choose whether to generate a termination function for a model by using the **Terminate function required** parameter. If your application runs indefinitely, clear this parameter. For more information, see Terminate function required.

To preview and customize the name of a generated C terminate entry-point function an Embedded Coder license is required. To preview the terminate entry-point function, open the Code Mappings editor and click the **Functions** tab. To customize the function name, in the **Function Name** column click and edit the spreadsheet directly. To customize the function using a template, in the **Function Customization Template** column select a template to apply to the function. For more information, see “Customize Generated Entry-Point C Function Names”, and “Configure Default Code Generation for Functions”.

To view the generated terminate entry-point function, open the **Code** view or Code Generation Report and examine the source code for your model. For more information see, “Analyze Generated Data Code Interface Report”.

Examples

Generate A Terminate Entry-Point Function

This example shows the basic workflow for how to configure, customize, generate, and examine the terminate entry-point function.

- 1 Open a model.
- 2 Select a coder. In the Apps gallery, click **Simulink Coder** or **Embedded Coder**.
- 3 Set model configuration parameters **Language** and **Code interface packaging**.
- 4 (Embedded Coder only) Customize the function. Using Embedded Coder, you can customize the name of the terminate entry-point function.
 - Open the Code Mappings editor.
 - Click the **Functions** tab.

- In the **Function Name** column, enter a name for the function.
- 5 Generate code.
 - 6 Examine the generated code. In the **Code** view, verify that the generated terminate function appears with the expected name and parameters.

Input Arguments

void — Default terminate function parameter

void (default)

The terminate entry-point C or C++ function provides an interface to terminate application code. By default, the generated function provides a **void-void** interface that does not have arguments.

Example: void

Output Arguments

void — Terminate function return value

void (default)

The terminate entry-point C or C++ function provides an interface to terminate application code. By default, the generated function provides a **void-void** interface that does not have a return value.

Example: void

Version History

Introduced before R2006a

See Also

`model_initialize` | `model_step` | `model_reset`

Topics

“Customize Generated Entry-Point C Function Names”

“Configure Generated C Function Interface for Model Entry-Point Functions”

“Configure Names for Individual C Entry-Point Functions”

“Configure Default Code Generation for Functions”

“Analyze Generated Data Code Interface Report”

“Startup, Reset, and Shutdown Function Interfaces”

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 ZIP file can include these types of files:

- Source files (for example, `.c`, `.cu`, and `.cpp` files)
- Header files (for example, `.h`, `.cuh`, 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.

- CMake configuration files (`CMakeLists.txt`) that you use to generate makefiles or projects for a compiler environment.

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.

To ensure that `packNGo` finds header files, add their paths to `buildInfo` by using the `addIncludePaths` function.

Note

- When generating standalone code by using the `codegen` command, you can use the `-package` option to both generate code and package the code in a ZIP file in a single step.
 - `packNGo` can modify the `buildInfo` argument. These modifications are reflected in further use of `buildInfo`, including in subsequent calls to `packNGo`.
-

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 build 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

You can configure ZIP file packaging by using the `set_param` function. During the build process, the code generator uses `packNGo` to create a ZIP file.

To configure ZIP file packaging for the model `zingbit`, run this command:

```
set_param('zingbit', 'PackageGeneratedCodeAndArtifacts', 'on');
```

The `packNGo` function creates the file `zingbit.zip`.

To specify another name for the ZIP file, for example, `portzingbit.zip`, run these commands:

```
set_param('zingbit', 'PackageGeneratedCodeAndArtifacts', 'on');  
set_param('zingbit', 'PackageName', 'portzingbit.zip')
```

Input Arguments

`buildInfo` — Build information object

RTW.BuildInfo object | path to `buildInfo.mat`

During the build process, the code generator places `buildInfo.mat` in the build folder. This MAT file contains the `RTW.BuildInfo` object. The object provides information that `packNGo` uses to produce the ZIP file.

You can specify the argument as the path to the `buildInfo.mat` file:

```
buildInfoFile = fullfile(pathToBuildFolder, 'buildInfo.mat');
packNGo(buildInfoFile, 'packType', 'hierarchical', ...
    'fileName', 'portzingbit');
```

Or, you can specify the argument as the path to the folder that contains `buildInfo.mat`:

```
packNGo(pathToBuildFolder, 'packType', 'hierarchical', ...
    'fileName', 'portzingbit');
```

You can also specify the argument as an `RTW.BuildInfo` object:

```
load buildInfo.mat
packNGo(buildInfo, 'packType', 'hierarchical', ...
    'fileName', 'portzingbit');
```

In situations where the `RTW.BuildInfo` object might not hold all the configuration information, use the path to the `buildInfo.mat` file or the folder that contains the file.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `'packType', 'flat', 'nestedZipFiles', true`

packType — Specify structure type of ZIP file

`'flat'` (default) | `'hierarchical'` | `'allHierarchical'`

If `'flat'`, package the generated code files in a ZIP file as a single, flat folder. The function checks whether files from separate folders have identical names, producing an error if there are duplicate filenames. The function does *not* package:

- Child `buildInfo.mat` files.
- `CMakeLists.txt` files.

If `'hierarchical'`, package the generated code files hierarchically in a primary ZIP file. The hierarchy contains top model, referenced model, and shared utility folders. The function also packages:

- The corresponding `buildInfo.mat` files for the folders.
- `CMakeLists.txt` files in the build folder.

In the `otherFiles` folder or `otherFiles.zip` file, the function flattens files that are not located beneath `matlabroot` or the code generation folder.

If `'allHierarchical'`, package the generated code files hierarchically in a primary ZIP file. In the `otherFiles` folder or `otherFiles.zip` file, the function preserves the folder hierarchy relative to the root of the file system.

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 located beneath *matlabroot*
- `sDirFiles.zip` — Files located beneath the code generation folder
- `otherFiles.zip` — Required files that are not located beneath *matlabroot* or the code generation folder

If `false`, create a primary ZIP file that contains folders, for example, the 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 — Option to include only minimally required header files

`true` (default) | `false`

Option to include only minimally required header files in the ZIP file, specified as `true` or `false`.

PackNGo uses the `findIncludeFiles` function with `minimalHeaders` to determine which header files to include, and updates `buildInfo` accordingly.

- If `minimalHeaders` is `true`, only the header files required to build the code are included.
- If `minimalHeaders` is `false`, all header files on the include paths specified by `buildInfo` are included.

Note GPU Coder™ requires that `minimalHeaders` is specified as `false`.

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` and `packType` is set to `'hierarchical'` or `'allHierarchical'`, 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`, `.cuh`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
- The function does not package source files for reusable library subsystems.
- 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.

Version History

Introduced in R2006b**R2025a: 'allHierarchical' Option for packType Name-Value Argument**

The `packType` name-value argument of the `packNGo` function provides the option `'allHierarchical'`. If you specify the option, the function packages the generated code files hierarchically in a primary ZIP file. In the `otherFiles` folder or `otherFiles.zip` file, the function preserves the folder hierarchy relative to the root of the file system.

See Also

`codebuild` | `addIncludeFiles` | `getIncludeFiles` | `findIncludeFiles`

Topics

“Customize Post-Code-Generation Build Processing”

“Relocate or Share Generated Code”

“Compile Code in Another Development Environment”

removeIncludePaths

Namespace: RTW

Remove include paths from build information

Syntax

```
removeIncludePaths(buildinfo, pathsToRemove)
removeIncludePaths(buildinfo, pathsToRemove, groups)
```

Description

`removeIncludePaths(buildinfo, pathsToRemove)` removes the specified include file paths from the build information.

`removeIncludePaths(buildinfo, pathsToRemove, groups)` removes the include file paths in the specified groups from the build information.

Examples

Remove include File Path From Build Information

Add the include path `/etcproj/etc/etc_build` to the build information object `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,...
    '/etcproj/etc/etc_build');
```

To remove the include path from the build information, run:

```
removeIncludePaths(myBuildInfo,...
    '/etcproj/etc/etc_build');
```

Remove Specific and Group of include File Paths From Build Information

Add the include paths `/etcproj/etc/lib`, `/etcproj/etc/etc_build`, `/common/lib1`, and `/common/lib2` to the build information object `myBuildInfo`. Associate the `etc` group with the paths `/etc/proj/etc/lib` and `/etcproj/etc/etc_build` and the `shared` group with the paths `/common/lib1` and `/common/lib2`.

```
myBuildInfo = RTW.BuildInfo;
addIncludePaths(myBuildInfo,...
    {'/etc/proj/etc/lib' '/etcproj/etc/etc_build' ...
     '/common/lib1' '/common/lib2'}, ...
    {'etc' 'etc' 'shared' 'shared'});
```

To remove `/etcproj/etc/lib` from the build information if the path is in the `shared` group, run:

```
removeIncludePaths(myBuildInfo, '/etcproj/etc/lib','shared');
```

Or, to remove the include file paths in both groups, run:

```
removeIncludePaths(myBuildInfo, '', {'etc' 'shared'});
```

Input Arguments

buildinfo — Build information object

object

RTW.BuildInfo object that contains information for compiling and linking generated code.

Example: myBuildInfo

pathsToRemove — include file paths to remove

character vector | array of character vectors | string

Specify include paths that you want to remove from the build information.

Example: '/proj/src'

groups — Groups of include file paths to remove

character vector | array of character vectors | string

Specify groups of include paths that you want to remove from the build information.

Example: {'etc' 'shared'}

Version History

Introduced in R2023b

See Also

`addIncludePaths`

Topics

“Customize Post-Code-Generation Build Processing”

removeSourceFiles

Remove source files from build information object

Syntax

```
removeSourceFiles(buildInfo,filenames,groups)
```

Description

`removeSourceFiles(buildInfo,filenames,groups)` removes source file names from build information that is stored in an `RTW.BuildInfo` object.

Examples

Remove Source File Names from Build Information

Suppose you add a list of source files to an `RTW.BuildInfo` object, `myBuildInfo`.

```
myBuildInfo = RTW.BuildInfo;
addSourceFiles(myBuildInfo, ...
    {'test1.c' 'test2.c' 'driver1.c' 'driver2.c'}, ...
    '/proj/src', ...
    {'Tests' 'Tests' 'Drivers' 'Drivers'});
```

To remove `test1.c` from the object, run:

```
removeSourceFiles(myBuildInfo, 'test1.c');
```

To remove `test2.c` and `driver1.c`, run:

```
removeSourceFiles(myBuildInfo, {'test2.c' 'driver1.c'});
```

To remove the names of files that belong to the `'Drivers'` group, run:

```
removeSourceFiles(myBuildInfo, '', 'Drivers');
```

Input Arguments

buildInfo — Build information object
object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

Example: `myBuildInfo`

filenames — Source files to remove
character vector | array of character vectors | string

Specify source files that you want to remove from the build information object.

Example: `'test1.c'`

groups — File groups to remove

character vector | array of character vectors | string

Specify groups of source files that you want to remove from the build information object.

Example: {'Tests' 'Drivers'}

Version History

Introduced in R2021b

See Also

`addSourceFiles`

Topics

“Customize Post-Code-Generation Build Processing”

removeUndefines

Namespace: RTW

Remove undefine preprocessor macros from build information

Syntax

```
removeUndefines(buildinfo, undefs)
```

Description

removeUndefines(buildinfo, undefs) removes undefine preprocessor macros from the build information.

Examples

Remove undefine Preprocessor Macros from Build Information

This example shows how to remove undefine preprocessor macros from a build information object.

Suppose you have two files, myFile.c and myInc.h, in your current working folder.

```
// myFile.c
#include <stdio.h>
int main(void) {
    #ifdef __STDC_VERSION__
        printf("__STDC_VERSION__ is defined\n");
    #else
        printf("__STDC_VERSION__ is not defined\n");
    #endif

    #ifdef MY_MACRO
        printf("MY_MACRO is defined\n");
    #else
        printf("MY_MACRO is not defined\n");
    #endif
}
```

```
// myInc.h
#ifndef MYINC_H
#define MYINC_H

#define MY_MACRO

#endif
```

Create an RTW.BuildInfo object and specify source files, undefine macros, and preinclude files.

```
buildInfo = RTW.BuildInfo;
buildInfo.ComponentName = 'MyComponent';
```

```
buildInfo.addSourceFiles('myfile.c');  
buildInfo.addUndefines('__STDC_VERSION__');  
buildInfo.addPreincludeFiles('myinc.h');
```

Get the undefine preprocessor macros.

```
undefines=getUndefines(buildInfo)
```

```
undefines =
```

```
1×1 cell array
```

```
{'__STDC_VERSION__'}
```

Remove the undefine preprocessor macro `__STDC_VERSION__`.

```
removeUndefines(buildInfo, {'__STDC_VERSION__'});
```

Check that the undefine preprocessor macro is removed.

```
undefines=getUndefines(buildInfo)
```

```
undefines =
```

```
0×0 empty cell array
```

Input Arguments

buildinfo — Build information object

object

Required. RTW.BuildInfo object that contains information for compiling and linking generated code.

undefs — undefine macros to remove

empty (default) | character vector | cell array of character vectors | string scalar

Required. Specify the undefine preprocessor macros that you want to remove from the build information object.

Version History

Introduced in R2024a

See Also

`addUndefines` | `getUndefines` | `addPreincludeFiles` | `getPreincludeFiles`

Topics

“Define Microsoft Visual Studio Toolchain for Building 32-Bit Application”

rtiostreamtest

Test custom `rtiostream` interface implementation

Syntax

```
rtiostreamtest(connection, parameterOne, parameterTwo, verbosityFlag)
rtiostreamtest('tcp', host, port)
rtiostreamtest('serial', port, baud)
```

Description

`rtiostreamtest(connection, parameterOne, parameterTwo, verbosityFlag)` runs a test suite to verify your custom `rtiostream` interface implementation.

`rtiostreamtest('tcp', host, port)`, via TCP/IP communication, connects MATLAB to the target hardware using the specified host and port.

`rtiostreamtest('serial', port, baud)`, via serial communication, connects MATLAB to the target hardware using the specified port and baud value.

During initialization, the function uses basic `rtiostream` I/O. The function determines:

- Byte ordering of data on the target hardware.
- Granularity of memory address.
- Size of data types.
- Whether `rtIOStreamRecv` blocks, that is, when there is no data whether `rtIOStreamRecv` waits for data or returns immediately with size received == 0.
- The size (`BUFFER_SIZE`) of its internal buffer for receiving or transmitting data through `rtiostream`. The default is 128 bytes.

In Test 1 (fixed size data exchange), the function:

- Checks data can be sent and received correctly in different chunk sizes. The chunk sizes for your development computer and target hardware are *symmetric*.
- Sends data as a known sequence that it can validate.
- Performs “host-to-target” tests. Your development computer sends data and your target hardware receives data in successive chunks of 1, 4, and 128 bytes.
- Performs “target-to-host” tests. Your target hardware sends data and your development computer receives data in successive chunks of 1, 4, and 128 bytes.

In Test 2 (varying size data exchange), the function:

- Checks that data can be sent and received correctly in different chunk sizes. The chunk sizes for your development computer and target hardware are *asymmetric*.
- Sends data as a known sequence that it can validate.
- Performs “host-to-target” tests:

- Your development computer sends data in chunks of 128 bytes and your target hardware receives data in chunks of 64 bytes.
- Your development computer sends data in chunks of 64 bytes and your target hardware receives data in chunks of 128 bytes.
- Performs “target-to-host” tests:
 - Your target hardware sends data in chunks of 64 bytes and your development computer receives data in chunks of 128 bytes.
 - Your target hardware sends data in chunks of 128 bytes and your development computer receives data in chunks of 64 bytes.

In **Test 3 (receive buffer detection)**, the function determines the data that it can store in between calls to `rtIOStreamRecv` on the target hardware. The function uses an iterative process:

- 1 The development computer transmits a data sequence while the target hardware sleeps. `rtIOStreamRecv` is not called while the target hardware sleeps.
- 2 When the target hardware wakes up, it calls `rtIOStreamRecv` to receive data from the internal buffer of the driver.
- 3 The function determines whether the internal buffer overflowed by checking for errors and checking the received data values.
- 4 If there are no overflow errors and the transmitted data is received correctly, the function starts another iteration, performing step 1 with a larger data sequence.

The function reports the size of the last known good buffer.

Examples

Verify Behavior of Custom `rtiostream` Interface Implementation

The test suite consists of two parts. One part of the test suite is an application that runs on the target hardware. The other part runs in MATLAB.

- 1 To create the target application, compile and link these files:
 - `rtiostreamtest.c`
 - `rtiostreamtest.h`
 - `rtiostream.h`
 - The `rtiostream` implementation under investigation, for example, `rtiostream_tcpip.c`.
 - `main.c`

`rtiostreamtest.c`, `rtiostreamtest.h`, and `main.c` are located in `matlabroot/toolbox/coder/rtiostream/src/rtiostreamtest`.

- 2 Download and run the application on your target hardware.
- 3 To run the MATLAB part of the test suite, invoke the `rtiostreamtest` function. For example:

```
rtiostreamtest('tcp','myProcessor','2345')
```

The function produces an output like this:

```
### Test suite for rtiostream ###
Initializing connection with target...
```

```

### Hardware characteristics discovered
Size of char   : 8 bit
Size of short  : 16 bit
Size of int    : 32 bit
Size of long   : 32 bit
Size of float  : 32 bit
Size of double : 64 bit
Size of pointer: 64 bit
Byte ordering  : Little Endian

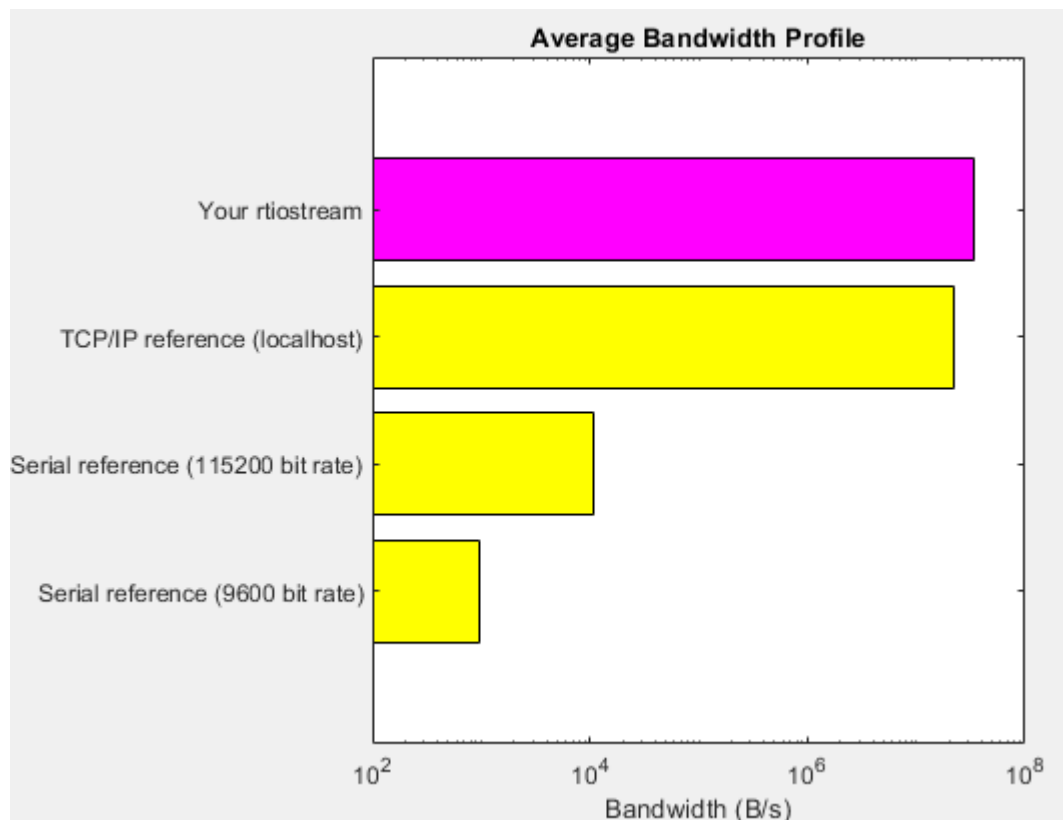
### rtiostream characteristics discovered
Round trip time : 0.25098 ms
rtIOStreamRecv behavior : non-blocking

### Test results
Test 1 (fixed size data exchange): ..... PASS
Test 2 (varying size data exchange): ..... PASS

### Test suite for rtiostream finished successfully ###

```

The function also generates the average bandwidth profile.



Input Arguments

connection — Transport protocol

'tcp' | 'serial'

Specify transport protocol for communication channel:

- 'tcp' -- TCP/IP
- 'serial' -- RS-232 serial

parameterOne — Host name or COM port

character vector | string scalar

If connection is 'tcp', specify name of target processor. For example, if your development computer is the target processor, you can specify 'localhost'.

If connection is 'serial', specify serial port ID, for example, 'COM1' for COM1, 'COM2' for COM2, and so on.

parameterTwo — Port number or baud value

integer

If connection is 'tcp', specify port number of TCP/IP server, an integer value between 256 and 65535.

If connection is 'serial', specify baud value, for example, 9600.

verbosityFlag — Verbosity

'' (default) | 'verbose'

If you specify 'verbose', the function displays messages that contain progress information. You can use the messages to debug runtime failures.

Version History

Introduced in R2013a

See Also

Topics

“Host-Target Communication for Simulink PIL Simulation”
“Host-Target Communication for MATLAB PIL Execution”
“Customize XCP Server Software”

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.

Examples

Return Global Parameter Structure

Return global parameter structure for model `rapidsim` to `param_struct`.

```
param_struct = rsimgetrtp('rapidsim')
param_struct =
    modelChecksum: [1.7165e+009 3.0726e+009 2.6061e+009
2.3064e+009]
    parameters: [1x1 struct]
```

Input Arguments

model — Model for which you are running rapid simulations

character vector | array of character vectors | string

Output Arguments

parameter_structure — Global parameter structure that includes tunable parameter information

parameter structure

The model parameter structure contains these fields:

Field	Description
<code>modelChecksum</code>	A four-element vector that encodes the structure. The code generator uses the <i>checksum</i> 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 <i>checksum</i> does not match the original <i>checksum</i> . The RSim executable detects this incompatibility in model parameter structures and exits to avoid returning incorrect simulation results. If the structure changes, regenerate code for the model.
<code>parameters</code>	A structure that defines model global parameters.

The `parameters` substructure includes these fields:

Field	Description
<code>dataTypeName</code>	Name of the parameter data type, for example, <code>double</code>
<code>dataTypeID</code>	An internal data type identifier
<code>complex</code>	Value 1 if parameter values are complex and 0 if real
<code>dtTransIdx</code>	Internal use only
<code>values</code>	Vector of parameter values
<code>structParamInfo</code>	Information about structure and bus parameters in the model

The `structParamInfo` substructure contains these fields:

Field	Description
<code>Identifier</code>	Name of the parameter
<code>ModelParam</code>	Value 1 if parameter is a model parameter and 0 if it is a block parameter
<code>BlockPath</code>	Block path for a block parameter and empty for model parameters
<code>CAPIDx</code>	Internal use only.

Do not modify fields in `structParamInfo`.

The function also includes an array of substructures, `map`, which represents tunable parameter information with these fields:

Field	Description
<code>Identifier</code>	Parameter name
<code>ValueIndices</code>	Vector of indices to parameter values
<code>Dimensions</code>	Vector indicating parameter dimensions

Version History

Introduced in R2006a

See Also

`rsimsetrtpparam`

Topics

"Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File"

"Tune Parameters"

"Author Block Algorithms"

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('model')`.

`rtP = rsimsetrtpparam(rtP,'paramName',paramValue)` takes an `rtP` structure that contains tunable parameter information and sets the values associated with `'paramName'` to be `paramValue`. You can specify multiple name-value pairs.

`rtP = rsimsetrtpparam(rtP,idx,'paramName',paramValue)` takes an `rtP` structure that contains tunable parameter information and sets the values associated with `'paramName'` to be `paramValue` in the `nth` `idx` parameter set. You can specify multiple name-value pairs. 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 — Parameter structure that contains sets of parameter names and respective values
parameter structure

idx — Index used to indicate the number of parameter sets in `rtP` structure
index of parameter sets

paramValue — Value of rtP parameter paramName

value of paramName

paramName — Name of parameter set to add to rtP structure

name of parameter set

Output Arguments

rtP — Expanded rtP parameter structure that contains the number of additional parameter sets indicated by **idx** and defined by the **rsimsetrtpparam** function call

expanded rtP parameter structure

Version History

Introduced in R2009b

See Also

[rsimgetrtP](#)

Topics

[“Create a MAT-File That Includes a Model Parameter Structure”](#)

[“Tune Parameters”](#)

[Default parameter behavior](#)

[“Author Block Algorithms”](#)

rtw_precompile_libs

Build model libraries 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. Use the function to precompile new or updated S-function libraries (MEX-files), which can reduce the overall build time for the model.

Model builds that use the template makefile approach support the function. Model builds that use the toolchain approach do not support the 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, 'TargetPreComplLibLocation',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

With the template makefile build approach, you can specify other libraries and how to build them. Use this field if you must precompile libraries.

Example: `build_spec.addLibs = 'libs_list';`

Version History

Introduced in R2009b

See Also

Topics

“Precompile S-Function Libraries”

“Approaches for Building Code Generated from Simulink Models”

“Use `rtwmakecfg.m` API to Customize Generated Makefiles”

rtwbuild

(Not recommended) Build generated code from a model

Note rtwbuild is not recommended. Use slbuild instead. rtwbuild is an alias for slbuild. If you run rtwbuild(varargin), the software executes slbuild(varargin).

Syntax

rtwbuild(varargin)

Description

rtwbuild(varargin) runs the command slbuild as shown by this code:

```
function varargout = rtwbuild(varargin)
    [varargout{1:nargout}] = slbuild(varargin{:});
end
```

Input Arguments

varargin — Argument list

cell array

Input argument list that rtwbuild passes to slbuild.

Version History

Introduced in R2009a

R2021a: rtwbuild is alias for slbuild

Behavior changed in R2021a

The rtwbuild function is an alias for the slbuild function. If you run rtwbuild(varargin), the software executes slbuild(varargin).

See Also

slbuild

RTW.BuildInfo

Provide information for compiling and linking generated code

Description

An `RTW.BuildInfo` object contains information for compiling and linking generated code.

Creation

Syntax

```
buildInformation = RTW.BuildInfo
```

Description

`buildInformation = RTW.BuildInfo` returns a build information object. You can use the object to specify information for compiling and linking generated code. For example:

- Compiler options
- Preprocessor identifier definitions
- Linker options
- Source files and paths
- Include files and paths
- Precompiled external libraries

Properties

ComponentName — Component name

character vector | string

Name of generated code component.

Object Functions

<code>addCompileFlags</code>	Add compiler options to build information
<code>addDefines</code>	Add preprocessor macro definitions to build information
<code>addIncludeFiles</code>	Add include files to build information
<code>addIncludePaths</code>	Add include paths to build information
<code>addLinkFlags</code>	Add link options to build information
<code>addLinkObjects</code>	Add link objects to build information
<code>addNonBuildFiles</code>	Add nonbuild-related files to build information
<code>addPreincludeFiles</code>	Add preinclude files to build information
<code>addSourceFiles</code>	Add source files to build information
<code>addSourcePaths</code>	Add source paths to build information
<code>addTMFTokens</code>	Add template makefile (TMF) tokens to build information

<code>addUndefines</code>	Add undefine preprocessor directives to build information
<code>findBuildArg</code>	Find a specific build argument in build information
<code>findIncludeFiles</code>	Find and add include (header) files to build information
<code>getBuildArgs</code>	Get build arguments from build information
<code>getCompileFlags</code>	Get compiler options from build information
<code>getDefines</code>	Get preprocessor macro definitions from build information
<code>getFullFileList</code>	Get list of files from build information
<code>getIncludeFiles</code>	Get include files from build information
<code>getIncludePaths</code>	Get include paths from build information
<code>getLinkFlags</code>	Get link options from build information
<code>getNonBuildFiles</code>	Get nonbuild-related files from build information
<code>getPreincludeFiles</code>	Get preinclude files from build information
<code>getSourceFiles</code>	Get source files from build information
<code>getSourcePaths</code>	Get source paths from build information
<code>getUndefines</code>	From build information, get preprocessor macros to undefine
<code>removeIncludePaths</code>	Remove include paths from build information
<code>removeSourceFiles</code>	Remove source files from build information object
<code>removeUndefines</code>	Remove undefine preprocessor macros from build information
<code>setTargetProvidesMain</code>	Disable inclusion of code generator provided (generated or static) main.c source file during build
<code>updateFilePathsAndExtensions</code>	Update files in build information with missing paths and file extensions
<code>updateFileSeparator</code>	Update file separator character for file lists in build information

Examples

Retrieve Build Information Object

When you build generated code, the build process stores an `RTW.BuildInfo` object in the `buildInfo.mat` file. To retrieve the object, from the code generation folder that contains the `buildInfo.mat` file, run:

```
bi=load('buildInfo.mat');
bi.buildInfo
```

ans =

BuildInfo with properties:

```

    ComponentName: 'slexAircraftExample'
         Viewer: []
          Tokens: [27x1 RTW.BuildInfoKeyValuePair]
        BuildArgs: [13x1 RTW.BuildInfoKeyValuePair]
         MakeVars: []
         MakeArgs: ''
TargetPreCompLibLoc: ''
TargetLibSuffix: ''
      ModelRefs: []
         SysLib: [1x1 RTW.BuildInfoModules]
           Maps: [1x1 struct]
        LinkObj: []
        Options: [1x1 RTW.BuildInfoOptions]
           Inc: [1x1 RTW.BuildInfoModules]
```

```
        Src: [1x1 RTW.BuildInfoModules]
        Other: [1x1 RTW.BuildInfoModules]
        Path: []
        Settings: [1x1 RTW.BuildInfoSettings]
        DisplayLabel: 'Build Info'
        Group: ''
```

The object contains build information.

Configure RTW.BuildInfo to Specify Code for Compilation

This example shows how to create an RTW.BuildInfo object and register source files.

Create an RTW.BuildInfo object.

```
buildInfo = RTW.BuildInfo;
```

Register source files.

```
buildInfo.ComponentName = 'foo1';
addSourceFiles(buildInfo, 'foo1.c');
```

Specify the build method and create a static library.

```
tmf = fullfile(tmffolder, 'ert_vcx64.tmf');
buildResult1 = codebuild(pwd, buildInfo, tmf)
```

Version History

Introduced in R2006a

See Also

codebuild

Topics

“Customize Post-Code-Generation Build Processing”

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'.

```
openExample('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

object | *'modelName'*

Model for which to get the build folder.

Example: `'sldemo_fuelsys'`

Output Arguments

folderStruct — Structure with build folder information

struct

Structure that provides build folder information through these character vector fields:

- **BuildDirectory** — Fully qualified path to build folder for model.
- **CacheFolder** — Simulation cache folder, which is the root folder for model build artifacts that the simulation uses.
- **CodeGenFolder** — Code generation folder, which is the root folder for generated production code.
- **RelativeBuildDir** — Build folder path relative to the code generation folder.
- **BuildDirSuffix** — Suffix appended to model name to create build folder.
- **ModelRefRelativeRootSimDir** — Root model reference target simulation folder, relative to the simulation cache folder.
- **ModelRefRelativeRootTgtDir** — Root model reference target build folder, relative to the code generation folder.
- **ModelRefRelativeBuildDir** — Model reference target build folder, relative to the code generation folder.

- `ModelRefRelativeSimDir` — Model reference target simulation folder, relative to the simulation cache folder.
- `ModelRefRelativeHdlDir` — Model reference target HDL folder, relative to the code generation folder.
- `ModelRefDirSuffix` — Suffix appended to system target file name to create model reference build folder.
- `SharedUtilsSimDir` — Shared utility folder for simulation, relative to the simulation cache folder.
- `SharedUtilsTgtDir` — Shared utility folder for code generation, relative to the code generation folder.

Example: `folderstruct = RTW.getBuildDir('MyModel')`

Version History

Introduced in R2008b

See Also

`slbuild`

Topics

“Working Folder”

“Manage Build Process Folders”

RTW.reservedIdentifiers

Get reserved identifiers for code generation

Syntax

```
[c, cpp] = RTW.reservedIdentifiers()
```

Description

[c, cpp] = RTW.reservedIdentifiers() returns structures that contain lists of reserved keywords. If your model uses a reserved keyword as an identifier, then the code generator attempts to replace that keyword when it generates identifiers in the code. Use the RTW.reservedIdentifiers() function to retrieve the list of keywords that the code generator attempts to replace.

Examples

Retrieve Reserved Identifiers

Retrieve reserved identifiers that the code generator attempts to replace.

```
[c, cpp] = RTW.reservedIdentifiers()
```

```
c =
```

```
    struct with fields:
```

```
        reservedWords: {103×1 cell}
        fileNamees: {29×1 cell}
        namesInLibraries: {1112×1 cell}
```

```
cpp =
```

```
    struct with fields:
```

```
        reservedWords: {42×1 cell}
        fileNamees: {85×1 cell}
        namesInLibraries: {1112×1 cell}
```

Output Arguments

c — C reserved identifiers

struct

C reserved identifiers, returned as a struct.

The struct contains these fields.

Field Name	Description
reservedWords	Keywords that the code generator reserves because the C language specification reserves them, returned as a cell array. The code generator attempts to replace these if the model uses them as identifiers.
fileNames	Keywords that the code generator reserves because they match the names of C standard library files, returned as a cell array. The MATLAB Coder code generator attempts to rename generated files that match these keywords. You can use keywords that appear in only this field as identifiers. The Simulink Coder and Embedded Coder code generators do not perform this check.
namesInLibraries	Keywords that the code generator reserves because they are used in commonly used libraries, returned as a cell array. The code generator attempts to replace these if the model uses them as identifiers. This list is shared between C and C++ and is returned in full in each struct.

cpp — C++ reserved identifiers

struct

C++ reserved identifiers, returned as a struct.

The struct contains these fields.

Field Name	Description
reservedWords	Keywords that the C++ code generator reserves because the C++ language specification reserves them, returned as a cell array. The code generator attempts to replace these if the model uses them as identifiers. This list does not include keywords that the C language specification reserves, but the C++ code generator attempts to replace those keywords as well.

Field Name	Description
fileNames	Keywords that the code generator reserves because they match the names of C++ standard library files, returned as a cell array. The MATLAB Coder code generator attempts to rename generated files that match these keywords. You can use keywords that appear in only this field as identifiers. The Simulink Coder and Embedded Coder code generators do not perform this check. This list does not include filenames from the C standard library, but the C++ code generator attempts to replace those filenames as well.
namesInLibraries	Keywords that the code generator reserves because they are used in commonly used libraries, returned as a cell array. The code generator attempts to replace these if the model uses them as identifiers. This list is shared between C and C++ and is returned in full in each <code>struct</code> .

Version History

Introduced in R2023b

See Also

Topics

"Reserved Keywords"

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 calls `codebuild`. If the current working folder is not the build folder, the function exits with an error.

`rtwrebuild` calls `codebuild` 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, `codebuild` recompiles code for all models in the hierarchy.

In Parallel Computing Toolbox™ commands, for example, a `parfor` or `spmd` loop, do not invoke `rtwbuild`, `rtwrebuild`, or `slbuild` commands that build models that are configured for parallel builds. For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models by Using Parallel Builds”.

`rtwrebuild(model)` assumes that the current working folder is one level above the build folder and calls `codebuild`. 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. 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

Call `codebuild` 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

```
rtwrebuild()
```

Rebuild Code from Parent Folder of Build Folder

When the current working folder is one level above the build folder, call `codebuild` to recompile code.

```
rtwrebuild('mymodel')
```

Rebuild Code from a Folder

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: 'CounterModel'

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')`

Version History

Introduced in R2009a

See Also

`codebuild` | `rtwbuild` | `slbuild`

Topics

"Rebuild a Model"

rtwreport

(To be removed) Create generated code report for model with Simulink Report Generator

Note `rtwreport` will be removed in a future release. Use `coder.report.generate` instead. For more information, see “Version History”.

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”. If the software cannot find the folder, an error occurs and code is not generated.

Input Arguments

model — Model name

string scalar | character vector

Model name for which the report is generated, specified as a character vector.

Data Types: string | char

folder — Build folder name

string scalar | 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.

Data Types: string | char

Version History

Introduced in R2007a

R2021a: rtwreport function will be removed

Warns starting in R2021a

The function `rtwreport` will be removed in a future release. Use `coder.report.generate` instead.

To update your code, change instances of the function name `rtwreport` to `coder.report.generate`. You do not need to change the input arguments.

Unlike the `rtwreport` function, the `coder.report.generate` function provides additional input options that you can use to configure the generated report. However, the `coder.report.generate` function does not include snapshots of the model or the block execution order list.

See Also

`coder.report.generate`

Topics

“Document Generated Code with Simulink Report Generator”

Import Generated Code (Simulink Report Generator)

“Working with Report Explorer” (MATLAB Report Generator)

Code Generation Summary (Simulink Report Generator)

rtwtrace

Trace a block to generated code in code generation report

Syntax

```
rtwtrace('blockpath')  
rtwtrace('Simulink_identifier')  
rtwtrace('blockpath', 'hdl')  
rtwtrace('blockpath', 'plc')
```

Description

`rtwtrace('blockpath')` opens a 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** on page 12-8 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 a 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.

`rtwtrace('blockpath', 'hdl')` opens a 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 a 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

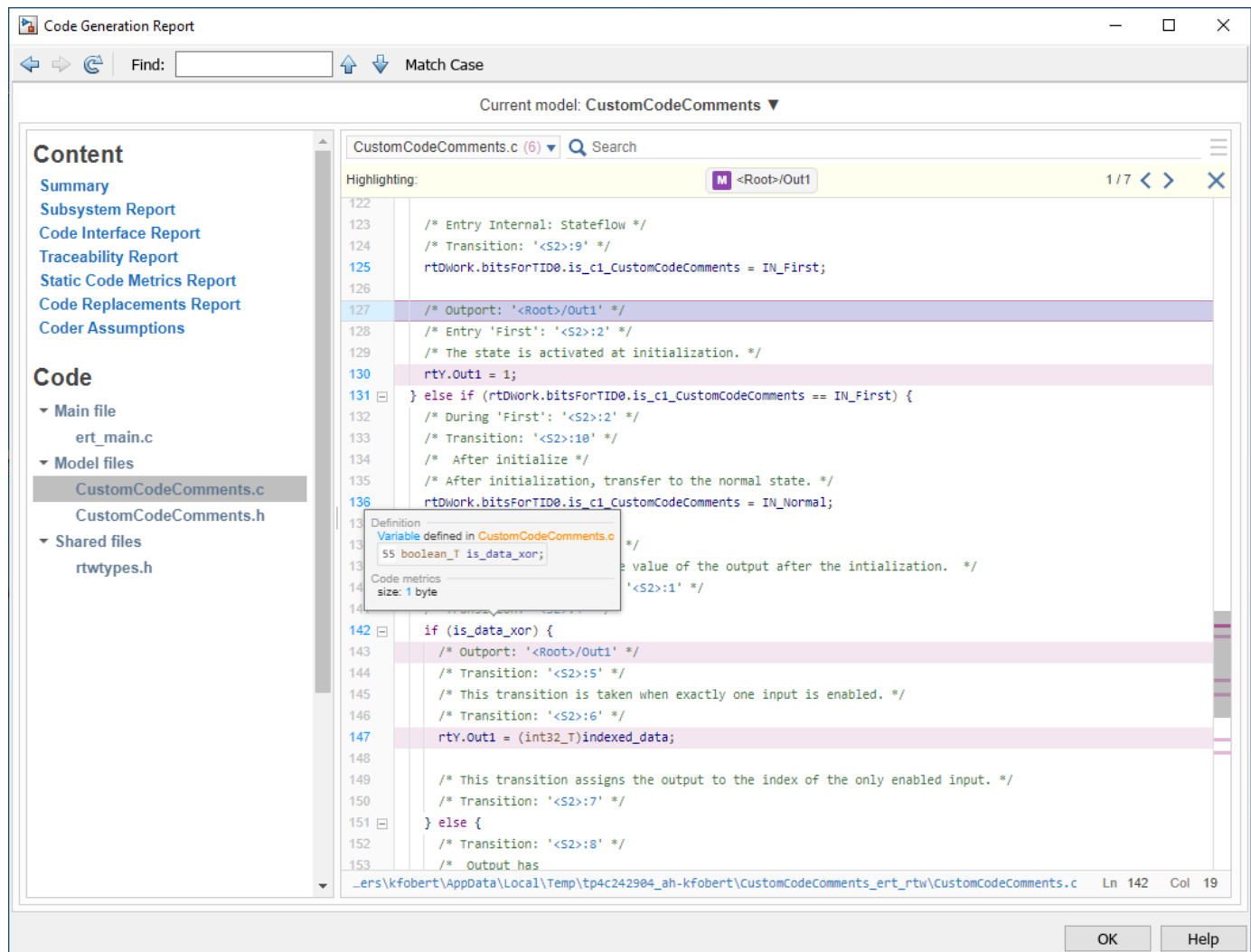
Display the generated code for a block in the `CustomCodeComments` model in code generation report

Open the `CustomCodeComments` model.

```
openExample('CustomCodeComments')
```

Use the block path to trace to the block in the generated code.

```
rtwtrace('CustomCodeComments/Out1')
```



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: '*CustomCodeComments/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*'.

For more information, see `Simulink.ID.getSID`.

Example: 'CustomCodeComments: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. To add the Embedded Coder app options to the menu, point to **Select Apps** and click the Embedded Coder button . Then, in the Embedded Coder app section, select **Navigate to C Code**.

Version History

Introduced in R2009b

See Also

Topics

"Verify Generated Code by Using Code Tracing"

"Model-to-Code Traceability"

Model-to-code

setTargetProvidesMain

Disable inclusion of code generator provided (generated or static) `main.c` source file during build

Syntax

```
setTargetProvidesMain(buildinfo,providesmain)
```

Description

`setTargetProvidesMain(buildinfo,providesmain)` disables the inclusion of a sample `main.c` source file from the code generator.

To replace the sample `main.c` file from the code generator with a custom `main.c` file:

- 1 In the `ert_make_rtw_hook.m` or `grt_make_rtw_hook.m` file, call the `setTargetProvidesMain` function at the 'after_tlc' stage.
- 2 If the **System target file** model configuration parameter is ERT-based, clear the **Generate an example main program** check box.

Examples

Workflow for setTargetProvidesMain

To apply the `setTargetProvidesMain` function:

Add `buildInfo` to the arguments in the function call.

```
function ert_make_rtw_hook(hookMethod,Name,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, Name, 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 project. When you indicate that the target provides `main.c`, the project requires this file to build without errors.

See also `addUserHook` Method to Customize Build Process.

Input Arguments

buildinfo — Build information object
object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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.

Version History

Introduced in R2015a

See Also

`addSourceFiles` | `addSourcePaths`

Topics

“Customize Build Process with STF_make_rtw_hook File”

“Customize Build Process with sl_customization.m”

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** – `CacheFolder`
- **Code generation folder** – `CodeGenFolder`
- **Code generation folder structure** – `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, ...
    'createDir', true, ...
    'keepPreviousPath', false);
```

```
'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.

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?” and “Files and Folders That MATLAB Accesses”.

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.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

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);`

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.

Version History

Introduced in R2010b

See Also

“Simulation cache folder” | “Code generation folder” | “Code generation folder structure”

Topics

“Manage Build Process Folders”

“Share Simulink Cache Files for Faster Simulation”

Simulink.ModelReference.modifyProtectedModel

Modify existing protected model

Syntax

```
harnessHandle = Simulink.ModelReference.modifyProtectedModel mdl
harnessHandle = Simulink.ModelReference.modifyProtectedModel mdl, Name=Value
[harnessHandle, neededVars] = Simulink.ModelReference.modifyProtectedModel( ____
)
```

Description

`harnessHandle = Simulink.ModelReference.modifyProtectedModel(mdl)` modifies options for an existing protected model specified by `mdl`. If you do not specify name-value arguments, the modified protected model is updated with default values and supports only simulation.

When model protection creates and opens a harness model, the handle of the harness model is returned by `harnessHandle`. By default, `harnessHandle` returns 0.

`harnessHandle = Simulink.ModelReference.modifyProtectedModel(mdl, Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes. For example, change the 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.

`[harnessHandle, neededVars] = Simulink.ModelReference.modifyProtectedModel( ____ )` returns the names of base workspace variables used by the protected model in `neededVars`. Specify an input argument combination from the previous syntaxes.

Examples

Update Protected Model with Default Values

Create a modifiable protected model with support for code generation, and then reset the protected model to use default values.

Open an example that contains the referenced model named `sldemo_mdref_counter`.

```
mdl = "sldemo_mdref_counter";
openExample(mdl)
```

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(...
mdl, "password");
```

Create a modifiable protected model with support for code generation.

```
Simulink.ModelReference.protect mdl, Mode="CodeGeneration", ...  
Modifiable=true, Report=true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
mdl, "password");
```

Modify the model to use default values.

```
Simulink.ModelReference.modifyProtectedModel(mdl);
```

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, and then modify the protected model to remove the web view support.

Open an example that contains the referenced model named `sldemo_mdref_counter`.

```
mdl = "sldemo_mdref_counter";  
openExample(mdl)
```

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(...  
mdl, "password");
```

Create a modifiable protected model with support for code generation and web view.

```
Simulink.ModelReference.protect(mdl, Mode="CodeGeneration", ...  
Webview=true, Modifiable=true, Report=true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
mdl, "password");
```

Remove support for web view from the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(mdl, ...  
Mode="CodeGeneration", Report=true);
```

Change Encryption Password for Code Generation

Change an encryption password for a modifiable protected model.

Open an example that contains the referenced model named `sldemo_mdref_counter`.

```
mdl = "sldemo_mdref_counter";  
openExample(mdl)
```

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(...  
mdl, "password");
```

Add the password that the protected model user must provide to generate code.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...  
mdl, "simpassword");
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect(mdl, Mode="CodeGeneration", ...  
Encrypt=true, Modifiable=true, Report=true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
mdl, "password");
```

Change the encryption password for simulation.

```
Simulink.ModelReference.modifyProtectedModel(mdl, ...  
Mode="CodeGeneration", Encrypt=true, Report=true, ...  
ChangeSimulationPassword={'simpassword', 'new_password'});
```

Determine Variables Required by Protected Model

Create a harness model for an existing protected model.

Open an example that contains the referenced model named CounterAlgorithm.

```
openExample('simulink/ComponentInterfaceSpecificationUsingBusesExample')
```

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.

```
mdl = "CounterAlgorithm";  
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
mdl, "password");
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect(mdl, Mode="CodeGeneration", ...  
Modifiable=true, Report=true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
mdl, "password");
```

Determine the variables required by the protected model.

```
[harness,neededVars] = Simulink.ModelReference.modifyProtectedModel(...  
mdl, Mode="CodeGeneration", Report=true);
```

The variables you must send the recipient are given by `neededVars`. The value of `neededVars` is a cell array that contains the names of the variables required by the protected model. However, the cell array can also contain the names of variables that the model does not need.

Tip To automatically package required variable definitions with the protected model in a project, set `Project` to `true`.

Before you share the protected model, edit `neededVars` to delete the names of any variables that the model does not need. Save the required variables in a data dictionary.

Input Arguments

mdl — Model name

character vector | string scalar

Model name, specified as a string or character vector. Specify the name of the protected model that you want to modify or the path of the Model block that references the protected model.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:

`Simulink.ModelReference.modifyProtectedModel mdl, Mode="CodeGeneration", OutputFormat="CompiledBinaries", ObfuscateCode=true` specifies that obfuscated code be generated for the protected model and that only binary files and headers in the generated code be visible to users of the protected model.

File Options

Project — Option to collect dependencies in project

`false` (default) | `true`

Option to collect dependencies in project, specified as `false` or `true`.

Specify whether to package supporting files, including a harness model, with the protected model in a project archive. The type and number of supporting files depends on the model being protected. Examples of supporting files are a MAT file with base workspace definitions and a data dictionary pruned to relevant definitions. The supporting files are not protected.

- `false` — Create only the protected model. If the protected model has dependencies, you must share them separately. Similarly, if you create a harness model for the protected model, you must share the harness model separately.
- `true` — Create a project archive (`.mlproj`) that contains the protected model, its dependencies, and its harness model. The supporting files are not protected. The project archive is a single file that allows for easy sharing. To create the interactive project, open the project archive.

Example: `Simulink.ModelReference.modifyProtectedModel mdl, Project=true`

Tips

Before sharing the project, check whether the project contains the required supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed. For more information, see “Manage Project Files”.

Dependencies

To set `Project` to `true`, set `Mode` to a value that supports simulation. For example, set `Mode` to "Simulation" (default).

Data Types: `logical`

ProjectName — Custom project name

character vector | string scalar

Custom project name, specified as a character vector or string scalar.

Specify the name of the project archive that contains the generated files. The project inside the archive uses the same name.

If you do not specify a custom project name, the default name for the project is the protected model name followed by `_protected`.

Example:

```
Simulink.ModelReference.modifyProtectedModel mdl, Project=true, ProjectName="My Project")
```

Dependencies

To enable this argument, set `Project` to `true`.

Data Types: `char` | `string`

Harness — Option to create harness model

`false` (default) | `true`

Option to create harness model, specified as a Boolean value.

Specify whether to create a harness model for the protected model. The harness model provides an isolated environment for the protected model, which is referenced by a Model block.

- `false` — Do not create a harness model for the protected model.
- `true` — Create a harness model for the protected model.

When you create a harness model for a protected model that relies on base workspace definitions, the software creates a MAT file that contains the base workspace definitions.

The harness model must have access to supporting files, such as a MAT file with base workspace definitions or a data dictionary.

Example: `Simulink.ModelReference.modifyProtectedModel mdl, Harness=true)`

Dependencies

- To set `Harness` to `true`, set `Mode` to a value that supports simulation. For example, set `Mode` to `"Simulation"` (default).
- When you set `Project` to `true`, `Harness` is set to `true`.

Data Types: `logical`

Path — Folder for protected model

current working folder (default) | character vector | string scalar

Folder for protected model, specified as a character vector or string scalar.

Example: `Simulink.ModelReference.modifyProtectedModel mdl, Path="C:\Work")`

Dependencies

The model being protected must be on the MATLAB path.

Data Types: `char` | `string`

Tag — Custom text string for additional information

character vector | string scalar

Custom text string for additional information, specified as a character vector or string scalar.

Being able to trace a protected model to the corresponding version of the design file can be useful for debugging. To store file provenance information in a protected model, specify a custom text tag with information such as the job number, Git™ commit, or continuous integration system tag associated with creating the protected model.

The tag length must not exceed 256 characters.

To generate a tag based on environment variables, use the `getenv` function.

Example: `Simulink.ModelReference.modifyProtectedModel mdl, Tag=myTag)`

Data Types: `char` | `string`

Verbose — Verbose output of build process information

`false` or `0` (default) | `true` or `1`

Verbose output of build process information, specified as a numeric or logical `1` (`true`) or `0` (`false`).

By default, the model protection process displays limited build process information. To display verbose build process information during model protection, set `Verbose` to `true`.

To control the display of build process information, the model protection process temporarily overrides the values of the **Verbose accelerator builds** and **Verbose build** configuration parameters.

Example: `Simulink.ModelReference.modifyProtectedModel mdl, Verbose=true)`

Data Types: `logical`

Functionality Options

Report — Option to generate report

false (default) | true

Option to generate report, specified as a Boolean value.

To view the report, perform one of these actions:

- Simulink Editor: On the Model block, right-click the protected-model badge icon . Then, select **Display Report**.
- MATLAB Command Window: Use the `Simulink.ProtectedModel.open` function. For example, to display the report for a protected model named `sldemo_mdref_counter`, enter this command.

```
Simulink.ProtectedModel.open("sldemo_mdref_counter","report");
```

The report is generated in HTML format. It includes information on the environment, functionality, and interface for the protected model.

Example: `Simulink.ModelReference.modifyProtectedModel mdl,Report=true)`

Dependencies

To set `Report` to `true`, set `Mode` to a value that supports simulation. For example, set `Mode` to "Simulation" (default).

Data Types: logical

Webview — Option to include read-only web view

false (default) | true

Option to include read-only web view, specified as a Boolean value.

Specify whether to share an interactive rendition of a Simulink model that the recipient can view in a web browser. This read-only web view of the model includes the contents and block parameters of the model.

- `false` — Do not share a web view of the model that you protect.
- `true` — Share a web view of the model that you protect.

To open a web view of a protected model, perform one of these actions:

- Files panel: Double-click the protected model file (.slxp).
- MATLAB Command Window: Use the `Simulink.ProtectedModel.open` function. For example, to display the web view for a protected model named `sldemo_mdref_counter`, enter this command.

```
Simulink.ProtectedModel.open("sldemo_mdref_counter","webview");
```

Example: `Simulink.ModelReference.modifyProtectedModel mdl,Webview=true)`

Dependencies

Creating a read-only web view of the model requires a Simulink Report Generator™ license.

Data Types: `logical`

Mode — Model protection mode

`"Simulation"` (default) | `"CodeGeneration"` | `"HDLCodeGeneration"` | `"ViewOnly"`

Model protection mode, specified as one of these values:

- `"Simulation"` — A model that references the protected model can run in normal, accelerator, or rapid accelerator mode.

Use `"Accelerator"` instead of `"Simulation"`.
- `"CodeGeneration"` — A model that references the protected model can run in normal, accelerator, rapid accelerator, or external mode and can support C and C++ code generation. This option requires a Simulink Coder license.
- `"HDLCodeGeneration"` — A model that references the protected model can run in normal, accelerator, or rapid accelerator mode and can support HDL code generation. This option requires an HDL Coder license.
- `"ViewOnly"` — The protected model supports only a read-only web view. The protected model does not support simulation or code generation. This option requires a Simulink Report Generator license.

For information about simulation modes in model hierarchies, see “Choose Simulation Modes for Model Hierarchies”.

Example: `Simulink.ModelReference.modifyProtectedModel mdl,Mode="Simulation")`

CodeInterface — Interface for generated code

`"Model reference"` (default) | `"Top model"`

Interface for generated code, specified as one of these values:

- `"Model reference"` — A Model block that references the protected model can support C and C++ code generation and simulation in accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode. For SIL and PIL simulations, the software deploys the generated code as part of a larger application that uses the referenced model. The generated artifacts match those generated by the `slbuild` function with the `"ModelReferenceCoderTarget"` build specification.
- `"Top model"` — A Model block that references the protected model can support simulation in accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode. For SIL and PIL simulations, the software deploys the generated code as a standalone application. The generated artifacts match those generated by the `slbuild` function with the `"StandaloneCoderTarget"` build specification.

SIL and PIL simulations can verify the protected model code.

Example: `Simulink.ModelReference.modifyProtectedModel mdl,CodeInterface="Top model")`

Dependencies

The **System target file** configuration parameter must be set to an ERT-based system target file, for example, `ert.tlc`. This option requires an Embedded Coder license.

hdl — Option to generate HDL code

`false` (default) | `true`

Option to generate HDL code, specified as a Boolean value.

For the protected model to support simulation, C and C++ code generation, and HDL code generation, set `Mode` to "CodeGeneration" and `hdl` to `true`.

Example:

```
Simulink.ModelReference.modifyProtectedModel mdl,Mode="CodeGeneration",hdl=true)
```

Tips

For the protected model to support simulation and HDL code generation, but not C or C++ code generation, set `Mode` to "HDLCodeGeneration". You do not have to set `hdl` to `true`.

Dependencies

- To generate HDL code, you must have an HDL Coder license.
- This argument affects the output only when you set `Mode` to "CodeGeneration".

Data Types: `logical`

Callbacks — Option to specify callbacks for protected model

`{}` (default) | cell array

Option to specify callbacks for protected model, specified as a cell array of `Simulink.ProtectedModel.Callback` objects.

Example:

```
Simulink.ModelReference.modifyProtectedModel mdl,Callbacks={pmcallback_sim,pmcallback_cg})
```

Data Types: `cell`

Protection Options

ObfuscateCode — Option to obfuscate generated code

`true` (default) | `false`

Option to obfuscate generated code, specified as a Boolean value.

Example:

```
Simulink.ModelReference.modifyProtectedModel mdl,Mode="CodeGeneration",ObfuscateCode=true)
```

Dependencies

This option is applied only when you set `Mode` to "CodeGeneration". Obfuscation is not supported for HDL code generation.

Data Types: `logical`

OutputFormat — Protected code visibility

"CompiledBinaries" (default) | "MinimalCode" | "AllReferencedHeaders"

Protected code visibility, specified as one of these values:

- "CompiledBinaries" — Include only compiled binaries for the generated code. The binary files and headers are visible.

- **"MinimalCode"** — Include only the minimal header files required to build the code with the chosen build settings. The code in the build folder is visible. Recipients can inspect the code in the protected model report and recompile it for their purposes.
- **"AllReferencedHeaders"** — Include header files found on the include path. The code in the build folder is visible. The headers referenced by the code are also visible.

This argument determines what part of the code generated for a protected model is visible to users.

Example:

```
Simulink.ModelReference.modifyProtectedModel mdl,Mode="CodeGeneration",Output  
Format="AllReferencedHeaders")
```

Dependencies

This argument affects the output only when you set **Mode** to "Simulation", "Accelerator", or "CodeGeneration".

Encrypt — Option to encrypt protected model

false (default) | true

Option to encrypt protected model, specified as a Boolean value.

This option applies when you have specified a password during protection, or by using these functions:

- 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`

To apply the specified passwords to the protected model, set **Encrypt** to **true**.

Example: `Simulink.ModelReference.modifyProtectedModel mdl,Encrypt=true)`

Data Types: logical

ChangeSimulationPassword — Option to change encryption password for simulation

cell array of two character vectors

Option to change 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:

```
Simulink.ModelReference.modifyProtectedModel mdl,Encrypt=true,ChangeSimulatio  
nPassword={'old_password','new_password'})
```

Dependencies

To enable this argument, set **Encrypt** to **true**.

Data Types: cell

ChangeViewPassword — Option to change encryption password for read-only view

cell array of two character vectors

Option to change 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:

```
Simulink.ModelReference.modifyProtectedModel mdl, Encrypt=true, ChangeViewPassword={'old_password', 'new_password'})
```

Dependencies

To enable this argument, set `Encrypt` to `true`.

Data Types: `cell`

ChangeCodeGenerationPassword — Option to change encryption password for code generation

cell array of two character vectors

Option to change 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:

```
Simulink.ModelReference.modifyProtectedModel mdl, Encrypt=true, ChangeCodeGenerationPassword={'old_password', 'new_password'})
```

Dependencies

To enable this argument, set `Encrypt` to `true`.

Data Types: `cell`

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 software calls the postprocessing function.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example:

```
Simulink.ModelReference.modifyProtectedModel mdl, Mode="CodeGeneration", CustomPostProcessingHook=@(protectedMdlInf)myHook(protectedMdlInf)
```

Dependencies

To enable this argument, set `Mode` to `"CodeGeneration"`.

Output Arguments

harnessHandle — Handle of harness model

double

Handle of harness model, returned as a double. When model protection does not create a harness model or packages the harness model in a project archive, the returned value is 0.

To create a harness model that is open after model protection, set `Harness` to `true` and `Project` to `false` (default).

neededVars — Names of base workspace variables

cell array

Names of base workspace variables that the protected model uses, returned as a cell array.

The cell array includes:

- Parameters that are used by the model interface
- Parameters that specify part of a data type, such as a bus object, enumerated type, or value type
- Parameters that are used by data store memory

The cell array can also include variables that the protected model does not use.

Version History

Introduced in R2014b

R2024b: Specify custom text tag with additional information

The new `Tag` argument lets you specify custom text with information that is relevant to the protected model.

Being able to trace a protected model to the corresponding version of the design file can be useful for debugging. To store file provenance information in a protected model, specify a custom text tag with information such as the job number, Git commit, or continuous integration system tag associated with creating the protected model.

R2024b: Model protection now provides limited build process information by default

Behavior changed in R2024b

To focus on the high-level steps in the model protection process, the `Simulink.ModelReference.modifyProtectedModel` function now displays limited build process information by default. The model protection process temporarily overrides the values of the **Verbose accelerator builds** and **Verbose build** configuration parameters.

To display verbose build process information during model protection, set `Verbose` to `true`.

R2024b: Protected model harness uses default instance parameter values

Behavior changed in R2024b

When you create a harness model for a protected model that has model arguments, the harness model uses the default instance parameter values for the Model block. Previously, the harness model inconsistently applied the instance parameter values.

R2023b: Collect protected model dependencies in project

When you modify a protected model, you can automatically package the protected model with its dependencies and a harness model in a project archive. When the recipient extracts the contents of the project archive and opens the harness model, they should be able to simulate the protected model without needing to define missing variables or objects. Before sharing the project archive, check whether the project contains the required supporting files, and update the harness model as needed.

Use `Project` to save the protected model, its dependencies, and its harness model in a project archive (.mlproj). The project archive provides a way to share a project in a single file.

Use `ProjectName` to specify a custom project name.

For more information, see “Package and Share Protected Models”

R2023b: External mode support with Mode set to 'CodeGeneration'

When you set `Mode` to 'CodeGeneration', the protected model supports external mode simulation.

R2023b: Default Mode value is 'Simulation'

Behavior changed in R2023b

The name of the default `Mode` value has changed from 'Accelerator' to 'Simulation'. The behavior of the default value has not changed.

R2023b: 'Accelerator' is not recommended

Not recommended starting in R2023b

The `Mode` value 'Accelerator' is not recommended. Use 'Simulation' instead.

- 'Simulation' and 'Accelerator' behave the same.
- 'Simulation' aligns better with the simulation option in the Create Protected Model dialog box.
- 'Accelerator' refers to only one of the supported simulation modes. A model that references a protected model that supports simulation can run in normal, accelerator, or rapid accelerator mode.

There are no plans to remove 'Accelerator'.

R2023a: Mode will not support normal mode restrictions

Behavior change in future release

In a future release, when you set `Mode` to 'Normal', the function will create a protected model that supports simulation when the parent model simulates in normal, accelerator, or rapid accelerator mode. This change will standardize simulation support for new protected models. Protected models created before this change will be unaffected.

Currently, when you set `Mode` to 'Normal', the function creates a protected model that supports simulation only when the parent model simulates in normal mode.

R2023a: Default Mode value is 'Accelerator'

Behavior changed in R2023a

The default value of the `Mode` argument is `'Accelerator'`. A model that references the protected model can run in normal, accelerator, or rapid accelerator mode.

With this change, the Create Protected Model dialog box and `Simulink.ModelReference.protect` function use the same default model protection mode.

Previously, the default value of the `Mode` argument was `'Normal'`. The `'Normal'` model protection mode restricts the use of the protected model such that the protected model supports simulation only when the parent model simulates in normal mode.

See Also

`Simulink.ModelReference.protect` |

`Simulink.ModelReference.ProtectedModel.setPasswordForModify`

Simulink.ModelReference.protect

Obscure referenced model contents to hide intellectual property

Syntax

```
harnessHandle = Simulink.ModelReference.protect mdl
harnessHandle = Simulink.ModelReference.protect mdl, Name=Value
[harnessHandle, neededVars] = Simulink.ModelReference.protect( ____ )
```

Description

`harnessHandle = Simulink.ModelReference.protect mdl` creates a protected model from the model specified by `mdl`. The protected model has the same name as the source model and an `.slxp` file extension. By default, the software places the protected model in the current working folder.

When model protection creates and opens a harness model, the handle of the harness model is returned by `harnessHandle`. By default, `harnessHandle` returns `0`.

`harnessHandle = Simulink.ModelReference.protect mdl, Name=Value` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes. For example, to collect the dependencies in a project, set `Project` to `true`.

`[harnessHandle, neededVars] = Simulink.ModelReference.protect( ____ )` returns the names of base workspace variables used by the protected model in `neededVars`. Specify an input argument combination from the previous syntaxes.

Examples

Protect Referenced Model

Open an example that contains the referenced model named `CounterAlgorithm`.

```
openExample('simulink/ComponentInterfaceSpecificationUsingBusesExample')
```

Protect the referenced model.

```
mdl = "CounterAlgorithm";
Simulink.ModelReference.protect(mdl);
```

The `Simulink.ModelReference.protect` function creates a protected model named `CounterAlgorithm.slxp` in the current working folder.

In the base workspace, the function returns `0` for the harness handle because the model protection process does not create a harness model by default.

When a Model block does not specify a file extension, the block refers to the protected model instead of the original model. By default, the model that references the protected model can run in normal, accelerator, or rapid accelerator mode.

Generate Code for Protected Model

When you generate code for a protected model, a model that references the protected model can support code generation.

For example, generate C and C++ code when you protect a referenced model named `CounterAlgorithm`.

Open an example that contains the referenced model.

```
openExample('simulink/ComponentInterfaceSpecificationUsingBusesExample')
```

Protect the referenced model, generate C and C++ code for the protected model, and do not obfuscate the generated code. Generating readable code for the protected model helps recipients integrate the protected model code with their own code.

```
mdl = "CounterAlgorithm";  
Simulink.ModelReference.protect(mdl, Mode="CodeGeneration", ...  
ObfuscateCode=false);
```

The function creates a protected model named `CounterAlgorithm.slxp`. A model that references the protected model can support C and C++ code generation.

For another example, generate HDL code when you protect a referenced model named `hdlcoder_referenced_model_gain`.

Open an example that contains the referenced model.

```
mdl = "hdlcoder_referenced_model_gain";  
openExample(mdl)
```

Protect the referenced model and generate HDL code for the protected model.

```
Simulink.ModelReference.protect(mdl, Mode="HDLCodeGeneration");
```

The function creates a protected model named `hdlcoder_referenced_model_gain.slxp`. A model that references the protected model can support HDL code generation.

To create a protected model that supports C, C++, and HDL code generation, set `Mode` to `"CodeGeneration"` and `hdl` to `true`.

Share Protected Model with Third Parties

When you plan to share a protected model with third parties, package the protected model, a harness model, and the supporting files in a project.

Open an example that contains the referenced model named `CounterAlgorithm`.

```
openExample('simulink/ComponentInterfaceSpecificationUsingBusesExample')
```


Protect the referenced model and collect the protected model dependencies in a project. Optionally, generate a report that describes the supported functionality.

```
mdl = "CounterAlgorithm";
Simulink.ModelReference.protect(mdl, Project=true, Report=true);
```

The function creates a protected model named `CounterAlgorithm.slxp` and a project archive named `CounterAlgorithm_protected.mlproj`.

The project archive contains:

- The protected model named `CounterAlgorithm.slxp`
- A harness model named `CounterAlgorithm_harness.slx`
- A data dictionary named `BusTypes.sldd`
- A Simulink cache file named `CounterAlgorithm_harness.slxc`

Before sharing the project, check whether the project contains the necessary supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify the missing files. Add the missing dependencies to the project and update the harness model as needed.

To share the protected model with a third party, share the project archive. For more information on how to create a project archive, see “Share Projects”.

For information about how recipients can use the protected model and project archive, see “Reference Protected Models from Third Parties”.

Tag Protected Model with Additional Information

The protected model automatically stores some information, such as the name and version of the design file and the platform used to create the protected model. When a continuous integration system creates the protected model, consider including provenance information such as the job number, Git commit, or continuous integration system tag associated with creating the protected model.

For example, suppose you use Jenkins® for continuous integration. Create a tag that contains provenance information for the protected model. For example, construct the tag using the `GIT_COMMIT` and `BUILD_ID` variables from the Jenkins environment.

```
JenkinsInfo = [getenv("GIT_COMMIT")+"@"+getenv("BUILD_ID")];
```

Protect the model and include the tag with the file provenance information. Optionally, generate a report that includes the model version used to generate the protected model. In this example, the model to protect is specified by `mdl`.

```
Simulink.ModelReference.protect(mdl, Tag=JenkinsInfo, Report=true);
```

The function creates a protected model. To get the information stored in the tag, open the protected model report or use the `slxpinfo` function.

Determine Variables Required by Protected Model

To simulate a model that references a protected model, you might need to define variables in the base workspace or data dictionaries. For example, the `CounterAlgorithm` model needs the variables that specify the buses at the root input and output ports of the model. When you ship a protected model, you must include definitions of the required variables or the model is unusable.

Tip To automatically package required variable definitions with the protected model in a project, set `Project` to `true`.

Open an example that contains the referenced model named `CounterAlgorithm`.

```
openExample('simulink/ComponentInterfaceSpecificationUsingBusesExample')
```

Generate the protected model, create a harness model for the protected model, and determine the variables required by the protected model.

```
mdl = "CounterAlgorithm";  
[harnessHandle,neededVars] = Simulink.ModelReference.protect(mdl, ...  
Harness=true)
```

The harness model handle is given by `harnessHandle`.

The variables you must send the recipient are given by `neededVars`. The value of `neededVars` is a cell array that contains the names of the variables required by the protected model. However, the cell array might also contain the names of variables that the model does not need.

Before you share the protected model, edit `neededVars` to delete the names of any variables that the model does not need. Save the required variables in a data dictionary.

Input Arguments

mdl — Model to protect

character vector | string scalar

Model to protect, specified as a character vector or string scalar. Specify the name of the model that you want to protect or the path of the Model block that references the model that you want to protect.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:

```
Simulink.ModelReference.protect(mdl,Mode="CodeGeneration",OutputFormat="CompiledBinaries",ObfuscateCode=true)
```

specifies that obfuscated code be generated for the protected model and that only binary files and headers in the generated code be visible to users of the protected model.

File Creation Options

Project — Option to collect dependencies in project

false (default) | true

Option to collect dependencies in project, specified as false or true.

Specify whether to package supporting files, including a harness model, with the protected model in a project archive. The type and number of supporting files depends on the model being protected. Examples of supporting files are a MAT file with base workspace definitions and a data dictionary pruned to relevant definitions. The supporting files are not protected.

- **false** — Create only the protected model. If the protected model has dependencies, you must share them separately. Similarly, if you create a harness model for the protected model, you must share the harness model separately.
- **true** — Create a project archive (.mlproj) that contains the protected model, its dependencies, and its harness model. The supporting files are not protected. The project archive is a single file that allows for easy sharing. To create the interactive project, open the project archive.

Example: `Simulink.ModelReference.protect mdl,Project=true)`

Tips

Before sharing the project, check whether the project contains the required supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed. For more information, see “Manage Project Files”.

Dependencies

To set Project to true, set Mode to a value that supports simulation. For example, set Mode to "Simulation" (default).

Data Types: logical

ProjectName — Custom project name

character vector | string scalar

Custom project name, specified as a character vector or string scalar.

Specify the name of the project archive that contains the generated files. The project inside the archive uses the same name.

If you do not specify a custom name, the default name for the project is the protected model name followed by _protected.

Example:

`Simulink.ModelReference.protect(mdl,Project=true,ProjectName="MyProject")`

Dependencies

To enable this argument, set Project to true.

Data Types: char | string

Harness — Option to create harness model

false (default) | true

Option to create harness model, specified as a Boolean value.

Specify whether to create a harness model for the protected model. The harness model provides an isolated environment for the protected model, which is referenced by a Model block.

- `false` — Do not create a harness model for the protected model.
- `true` — Create a harness model for the protected model.

When you create a harness model for a protected model that relies on base workspace definitions, the software creates a MAT file that contains the base workspace definitions.

The harness model must have access to supporting files, such as a MAT file with base workspace definitions or a data dictionary.

Example: `Simulink.ModelReference.protect mdl, Harness=true`

Dependencies

- To set `Harness` to `true`, set `Mode` to a value that supports simulation. For example, set `Mode` to `"Simulation"` (default).
- When you set `Project` to `true`, `Harness` is set to `true`.

Data Types: `logical`

Path — Folder for protected model

`current working folder (default) | character vector | string scalar`

Folder for protected model, specified as a character vector or string scalar.

Example: `Simulink.ModelReference.protect mdl, Path="C:\Work"`

Dependencies

The model being protected must be on the MATLAB path.

Data Types: `char | string`

Tag — Custom text string for additional information

`character vector | string scalar`

Custom text string for additional information, specified as a character vector or string scalar.

Being able to trace a protected model to the corresponding version of the design file can be useful for debugging. To store file provenance information in a protected model, specify a custom text tag with information such as the job number, Git commit, or continuous integration system tag associated with creating the protected model.

The tag length must not exceed 256 characters.

To generate a tag based on environment variables, use the `getenv` function.

Example: `Simulink.ModelReference.protect mdl, Tag=myTag`

Data Types: `char | string`

Verbose — Verbose output of build process information

`false` or `0` (default) | `true` or `1`

Verbose output of build process information, specified as a numeric or logical 1 (`true`) or 0 (`false`).

By default, the model protection process displays limited build process information. To display verbose build process information during model protection, set `Verbose` to `true`.

To control the display of build process information, the model protection process temporarily overrides the values of the **Verbose accelerator builds** and **Verbose build** configuration parameters.

Example: `Simulink.ModelReference.protect mdl, Verbose=true)`

Data Types: `logical`

Functionality Options

Report — Option to generate report

`false` (default) | `true`

Option to generate report, specified as a Boolean value.

To view the report, perform one of these actions:

- Simulink Editor: On the Model block, right-click the protected-model badge icon . Then, select **Display Report**.
- MATLAB Command Window: Use the `Simulink.ProtectedModel.open` function. For example, to display the report for a protected model specified by `mdl`, enter this command.

```
Simulink.ProtectedModel.open(mdl, "report");
```

The report is generated in HTML format. It includes information on the environment, functionality, and interface for the protected model.

Example: `Simulink.ModelReference.protect(mdl, Report=true)`

Dependencies

To set `Report` to `true`, set `Mode` to a value that supports simulation. For example, set `Mode` to `"Simulation"` (default).

Data Types: `logical`

Webview — Option to include read-only web view

`false` (default) | `true`

Option to include read-only web view, specified as a Boolean value.

Specify whether to share an interactive rendition of a Simulink model that the recipient can view in a web browser. This read-only web view of the model includes the contents and block parameters of the model.

- `false` — Do not share a web view of the model that you protect.
- `true` — Share a web view of the model that you protect.

To open a web view of a protected model, perform one of these actions:

- Files panel: Double-click the protected model file (`.slxp`).

- MATLAB Command Window: Use the `Simulink.ProtectedModel.open` function. For example, to display the web view for a protected model specified by `mdl`, enter this command.

```
Simulink.ProtectedModel.open(mdl,"webview");
```

```
Example: Simulink.ModelReference.protect(mdl,Webview=true)
```

Dependencies

Creating a read-only web view of the model requires a Simulink Report Generator license.

Data Types: `logical`

Mode — Model protection mode

"Simulation" (default) | "CodeGeneration" | "HDLCodeGeneration" | "ViewOnly"

Model protection mode, specified as one of these values:

- "Simulation" — A model that references the protected model can run in normal, accelerator, or rapid accelerator mode.

Use "Accelerator" instead of "Simulation".
- "CodeGeneration" — A model that references the protected model can run in normal, accelerator, rapid accelerator, or external mode and can support C and C++ code generation. This option requires a Simulink Coder license.
- "HDLCodeGeneration" — A model that references the protected model can run in normal, accelerator, or rapid accelerator mode and can support HDL code generation. This option requires an HDL Coder license.
- "ViewOnly" — The protected model supports only a read-only web view. The protected model does not support simulation or code generation. This option requires a Simulink Report Generator license.

For information about simulation modes in model hierarchies, see "Choose Simulation Modes for Model Hierarchies".

```
Example: Simulink.ModelReference.protect(mdl,Mode="Simulation")
```

CodeInterface — Interface for generated code

"Model reference" (default) | "Top model"

Interface for generated code, specified as one of these values:

- "Model reference" — A Model block that references the protected model can support C and C++ code generation and simulation in accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode. For SIL and PIL simulations, the software deploys the generated code as part of a larger application that uses the referenced model. The generated artifacts match those generated by the `slbuild` function with the "ModelReferenceCoderTarget" build specification.
- "Top model" — A Model block that references the protected model can support simulation in accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode. For SIL and PIL simulations, the software deploys the generated code as a standalone application. The generated artifacts match those generated by the `slbuild` function with the "StandaloneCoderTarget" build specification.

SIL and PIL simulations can verify the protected model code.

Example: `Simulink.ModelReference.protect mdl, CodeInterface="Top model")`

Dependencies

The **System target file** configuration parameter must be set to an ERT-based system target file, for example, `ert.tlc`. This option requires an Embedded Coder license.

hdl — Option to generate HDL code

`false` (default) | `true`

Option to generate HDL code, specified as a Boolean value.

For the protected model to support simulation, C and C++ code generation, and HDL code generation, set `Mode` to `"CodeGeneration"` and `hdl` to `true`.

Example: `Simulink.ModelReference.protect mdl, Mode="CodeGeneration", hdl=true)`

Tips

For the protected model to support simulation and HDL code generation, but not C or C++ code generation, set `Mode` to `"HDLCodeGeneration"`. You do not have to set `hdl` to `true`.

Dependencies

- To generate HDL code, you must have an HDL Coder license.
- This argument affects the output only when you set `Mode` to `"CodeGeneration"`.

Data Types: `logical`

TunableParameters — Tunable parameters for simulation

`"None"` or `{}` (default) | `"All"` | string array | cell array of character vectors

Tunable parameters for simulation, specified as `"None"`, `{}`, `"All"`, a string array, or a cell array of character vectors.

Specify the parameters from external data sources that you want the recipient to be able to tune during simulation.

The model protection process honors the tunability of parameters in the model workspace. Model arguments are inherently tunable. You do not need to specify the tunability of model arguments.

When you protect a model that references one or more protected models, specify the tunable parameters of each referenced protected model as also being tunable for the parent model. To get the list of tunable parameters for a protected model, use the `Simulink.ProtectedModel.getTunableParameters` function.

For more information about tunable parameters for simulation, see “Tune and Experiment with Block Parameter Values”.

Example:

`Simulink.ModelReference.protect mdl, TunableParameters=["param1", "param2"])`

Data Types: `char` | `string` | `cell`

Callbacks — Option to specify callbacks for protected model

`{}` (default) | cell array

Option to specify callbacks for protected model, specified as a cell array of `Simulink.ProtectedModel.Callback` objects.

Example:

```
Simulink.ModelReference.protect mdl, Callbacks={pmcallback_sim,pmcallback_cg})
```

Data Types: cell

Protection Options

ObfuscateCode — Option to obfuscate generated code

true (default) | false

Option to obfuscate generated code, specified as a Boolean value.

Example:

```
Simulink.ModelReference.protect mdl, Mode="CodeGeneration", ObfuscateCode=true)
```

Dependencies

This option is applied only when you set `Mode` to "CodeGeneration". Obfuscation is not supported for HDL code generation.

Data Types: logical

OutputFormat — Protected code visibility

"CompiledBinaries" (default) | "MinimalCode" | "AllReferencedHeaders"

Protected code visibility, specified as one of these values:

- "CompiledBinaries" — Include only compiled binaries for the generated code. The binary files and headers are visible.
- "MinimalCode" — Include only the minimal header files required to build the code with the chosen build settings. The code in the build folder is visible. Recipients can inspect the code in the protected model report and recompile it for their purposes.
- "AllReferencedHeaders" — Include header files found on the include path. The code in the build folder is visible. The headers referenced by the code are also visible.

This argument determines what part of the code generated for a protected model is visible to users.

Example:

```
Simulink.ModelReference.protect mdl, Mode="CodeGeneration", OutputFormat="AllReferencedHeaders")
```

Dependencies

This argument affects the output only when you set `Mode` to "Simulation", "Accelerator", or "CodeGeneration".

Encrypt — Option to encrypt protected model

false (default) | true

Option to encrypt protected model, specified as a Boolean value.

This option applies when you have specified a password during protection, or by using these functions:

- 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`

To apply the specified passwords to the protected model, set `Encrypt` to `true`.

Example: `Simulink.ModelReference.protect mdl,Encrypt=true)`

Data Types: `logical`

Sign — Option to sign protected model with digital certificate

character vector | string scalar

Option to sign protected model with digital certificate, specified as a character vector or string scalar that specifies the digital certificate. If the certificate file is password protected, use the `Simulink.ModelReference.ProtectedModel.setPasswordForCertificate` function to provide the password before you use the certificate.

Example: `Simulink.ModelReference.protect mdl,Sign="certificate_file.pfx")`

Data Types: `char` | `string`

Modifiable — Option to create modifiable protected model

`false` (default) | `true`

Option to create modifiable protected model, specified as a Boolean value.

To use this option:

- Add a password for modification by 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: `Simulink.ModelReference.protect mdl,Modifiable=true)`

Data Types: `logical`

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 software calls the postprocessing function.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example:

```
Simulink.ModelReference.protect mdl, Mode="CodeGeneration", CustomPostProcessingHook=@(protectedMdlInf)myHook(protectedMdlInf)
```

Dependencies

To enable this argument, set `Mode` to `"CodeGeneration"`.

Output Arguments

harnessHandle — Handle of harness model

double

Handle of harness model, returned as a double. When model protection does not create a harness model or packages the harness model in a project archive, the returned value is 0.

To create a harness model that is open after model protection, set `Harness` to `true` and `Project` to `false` (default).

neededVars — Names of base workspace variables

cell array

Names of base workspace variables that the protected model uses, returned as a cell array.

The cell array includes:

- Tunable parameters
- Parameters that are used by the model interface
- Parameters that specify part of a data type, such as a bus object, enumerated type, or value type
- Parameters that are used by data store memory

The cell array can also include variables that the protected model does not use.

Alternatives

"Protect Models to Conceal Contents"

Version History

Introduced in R2012b

R2024b: Specify custom text tag with additional information

The new `Tag` argument lets you specify custom text with information that is relevant to the protected model.

Being able to trace a protected model to the corresponding version of the design file can be useful for debugging. To store file provenance information in a protected model, specify a custom text tag with information such as the job number, Git commit, or continuous integration system tag associated with creating the protected model.

R2024b: Model protection now provides limited build process information by default

Behavior changed in R2024b

To focus on the high-level steps in the model protection process, the `Simulink.ModelReference.protect` function now displays limited build process information by default. The model protection process temporarily overrides the values of the **Verbose accelerator builds** and **Verbose build** configuration parameters.

To display verbose build process information during model protection, set `Verbose` to `true`.

R2024b: Protected model harness uses default instance parameter values

Behavior changed in R2024b

When you create a harness model for a protected model that has model arguments, the harness model uses the default instance parameter values for the Model block. Previously, the harness model inconsistently applied the instance parameter values.

R2023b: External mode support with Mode set to 'CodeGeneration'

When you set `Mode` to `'CodeGeneration'`, the protected model supports external mode simulation.

R2023b: Default Mode value is 'Simulation'

Behavior changed in R2023b

The name of the default `Mode` value has changed from `'Accelerator'` to `'Simulation'`. The behavior of the default value has not changed.

R2023b: Setting Mode to 'Accelerator' is not recommended

Not recommended starting in R2023b

Setting `Mode` to `'Accelerator'` is not recommended. Set `Mode` to `'Simulation'` instead.

- `'Simulation'` and `'Accelerator'` behave the same.
- `'Simulation'` aligns better with the simulation option in the Create Protected Model dialog box.
- `'Accelerator'` refers to only one of the supported simulation modes. A model that references a protected model that supports simulation can run in normal, accelerator, or rapid accelerator mode.

There are no plans to remove `'Accelerator'`.

R2023b: Setting Mode to 'Normal' is not recommended

Not recommended starting in R2023b

Setting `Mode` to `'Normal'` is not recommended. Set `Mode` to `'Simulation'` instead.

- When you set `Mode` to `'Normal'`, the function creates a protected model that supports simulation only when the parent model simulates in normal mode.
- `'Simulation'` aligns better with the simulation option in the Create Protected Model dialog box. When you set `Mode` to `'Simulation'`, the function creates a protected model that supports simulation when the parent model simulates in normal, accelerator, or rapid accelerator mode.

There are no plans to remove `'Normal'`.

R2023a: Default Mode value is 'Accelerator'

Behavior changed in R2023a

The default value of the `Mode` argument is `'Accelerator'`. A model that references the protected model can run in normal, accelerator, or rapid accelerator mode.

With this change, the Create Protected Model dialog box and `Simulink.ModelReference.protect` function use the same default model protection mode.

Previously, the default value of the `Mode` argument was `'Normal'`. The `'Normal'` model protection mode restricts the use of the protected model such that the protected model supports simulation only when the parent model simulates in normal mode.

R2022a: Tunable parameters for simulation are specified by the TunableParameters argument

Behavior changed in R2022a

You must specify the parameters of the protected model that you want to be tunable during simulation by using the `TunableParameters` name-value argument. By default, no parameters are tunable during simulation.

To get the list of tunable parameters for a protected model, use the `Simulink.ProtectedModel.getTunableParameters` function.

Previously, a parameter was tunable when its code generation storage class was set to a value other than `Auto`. By default, the storage class for an individual data element is `Auto`.

R2020b: Automatically package protected models with their dependencies

When you create a protected model, you can automatically package it with its dependencies and a harness model in a project archive. When the recipient extracts the contents of the project archive and opens the harness model, they should be able to simulate the protected model without needing to define missing variables or objects. Before sharing the project archive, check whether the project contains the required supporting files, and update the harness model as needed.

Use `Project` to save the protected model, its dependencies, and its harness model in a project archive (`.mlproj`). The project archive provides a way to share a project in a single file.

Use `ProjectName` to specify a custom project name.

For more information, see “Package and Share Protected Models”

See Also

Tools

Protected Model Creator

Functions

`Simulink.ModelReference.modifyProtectedModel` |
`Simulink.ModelReference.ProtectedModel.setPasswordForModify` |
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` |
`Simulink.ModelReference.ProtectedModel.setPasswordForView` |
`Simulink.ModelReference.ProtectedModel.clearPasswords` |
`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel` |
`Simulink.ProtectedModel.getTunableParameters`

Topics

“Protect Models to Conceal Contents”
“Protected Model Capabilities”
“Test Protected Models”
“Package and Share Protected Models”
“Specify Custom Obfuscators for Protected Models”
“Configure and Run SIL Simulation”
“Code Interfaces for SIL and PIL”
“Define Callbacks for Protected Models”
“Reference Protected Models from Third Parties”

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()
```

Version History

Introduced in R2014b

See Also

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel
```

Topics

“Protect Models to Conceal Contents”

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: 'CounterModel'

Data Types: char

Version History

Introduced in R2014b

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords`

Topics

"Protect Models to Conceal Contents"

Simulink.ProtectedModel.getTunableParameters

Get list of tunable parameters for protected model

Syntax

```
parameters = Simulink.ProtectedModel.getTunableParameters(model)
```

Description

`parameters = Simulink.ProtectedModel.getTunableParameters(model)` returns the tunable parameters for the protected model specified by `model`.

Examples

Get Tunable Parameters for Protected Model

Suppose you want to protect a model named `mymodel` and you want the recipient of the protected model to be able to tune parameter values during simulation. You can use the `Simulink.ModelReference.protect` function to protect the model with all tunable parameters available to the protected model recipient.

```
Simulink.ModelReference.protect("mymodel",TunableParameters="All")
```

The `Simulink.ProtectedModel.getTunableParameters` function returns the tunable parameters of the protected model.

```
Simulink.ProtectedModel.getTunableParameters("mymodel")
```

```
ans =  
    1×2 string array  
    "K"    "K2"
```

Input Arguments

model — Protected model to query

string scalar | character vector

Protected model to query, specified as a string scalar or character vector.

Data Types: `char` | `string`

Output Arguments

parameters — Tunable parameters in specified protected model

string array

Tunable parameters in specified protected model, returned as a string array.

Version History

Introduced in R2022a

See Also

`Simulink.ModelReference.protect`

Topics

"Protect Models to Conceal Contents"

"Test Protected Models"

"Package and Share Protected Models"

"Best Practices for Creating Protected Models"

Simulink.ModelReference.ProtectedModel.HookInfo

Files and exported symbols generated by creation of protected model

Description

Information about files and symbols generated when creating a protected model. You can use this information for postprocessing of the generated files prior to packaging. To access the properties of this class, create a postprocessing function that accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as input. When you protect your model, use the 'CustomPostProcessingHook' option of the `Simulink.ModelReference.protect` function to specify the postprocessing function. Prior to packaging the protected model, the postprocessing function is called with the `Simulink.ModelReference.ProtectedModel.HookInfo` from the protected model as input.

Properties

ExportedSymbols — Exported Symbols

cell array of character vectors

A list of exported symbols generated by a protected model that you must not modify, returned as a cell array of character vectors.

For a protected model with a top model interface, the `HookInfo` object cannot provide information on exported symbols.

NonSourceFiles — Nonsource code files

cell array of character vectors

A list of nonsource files generated by protected model creation, returned as a cell array of character vectors. Nonsource files include MAT, RSP, and PRJ files.

SourceFiles — Source code files

cell array of character vectors

A list of source code files generated by protected model creation, returned as a cell array of character vectors. Source files include C, H, CPP, and HPP files.

Examples

Use Protected Model Information in Postprocessing Function

- 1 On the MATLAB path, create a postprocessing function `pm_postprocessing.m` that contains this code:

```
function pm_postprocessing(hookInfoObject)
```

```

s1 = 'Exported Symbols: ';
symbols = hookInfoObject.ExportedSymbols;
s2 = 'Source Files: ';
srcfiles = hookInfoObject.SourceFiles;
s3 = 'Non-Source Files: ';
nonsrcfiles = hookInfoObject.NonSourceFiles;
disp([s1 symbols])
disp([s2 srcfiles])
disp([s3 nonsrcfiles])

```

This function displays a list of the exported symbols, source code files, and nonsource files generated by the model protection process.

- 2 Protect the model `sldemo_mdref_counter` and specify the postprocessing function that you created. Before packaging the generated files, the model protection process calls the postprocessing function and inputs the `Simulink.ModelReference.ProtectedModel.HookInfo` object that was generated for the protected model.

```

openExample('sldemo_mdref_counter');
Simulink.ModelReference.protect('sldemo_mdref_counter',...
'Mode', 'CodeGeneration',...
'CustomPostProcessingHook',...
@(protectedMdlInf)pm_postprocessing(protectedMdlInf))

```

Version History

Introduced in R2014a

See Also

`Simulink.ModelReference.protect`

Topics

“Specify Custom Obfuscators for Protected Models”

Simulink.ModelReference.ProtectedModel.setPasswordForCertificate

Provide password for digital certificate

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForCertificate(  
certificateFile,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForCertificate(certificateFile,password)` provides the required password to access the certificate file to digitally sign a protected model.

Examples

Sign a Protected Model by Using Password Protected Certificate

Protect a model, and then digitally sign it by using a password protected certificate.

Open and protect the model that you want to sign. For this example, protect the model `sldemo_mdref_counter`.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.protect('sldemo_mdref_counter');
```

Locate the certificate file that you want to use to sign the protected model. Enter the password for the certificate.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCertificate('certificate_file.pfx','password');
```

Sign the protected model by using the certificate file.

```
Simulink.ProtectedModel.sign('sldemo_mdref_counter.slxp','certificate_file.pfx');
```

Input Arguments

certificateFile — Certificate file to use for signing

character vector | string scalar

Certificate file to use for signing the protected model, specified as a character vector or string scalar. The certificate must be a PKCS #12 file with the extension `.pfx` or `.p12`.

Example: `'my_cert.pfx'`

Example: `'InstitutionCertificate.p12'`

password — Password for certificate file

string or character vector

Password, specified as a string or character vector. If the certificate is encrypted for code generation, the password is required.

Version History

Introduced in R2020a

See Also

`Simulink.ProtectedModel.sign`

Topics

“Sign a Protected Model”

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.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...  
'sldemo_mdref_counter', 'password');  
Simulink.ModelReference.protect('sldemo_mdref_counter', ...  
'Mode', 'CodeGeneration', 'Encrypt', true, 'Report', true);
```

A protected model named `sldemo_mdref_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.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...  
'sldemo_mdref_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.

Version History

Introduced in R2014b

See Also

Simulink.ModelReference.protect |
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation |
Simulink.ModelReference.ProtectedModel.setPasswordForView |
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration

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.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdref_counter','Mode',...  
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter','password');
```

Add support for Web view to the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...  
'sldemo_mdref_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.

Version History

Introduced in R2014b

See Also

`Simulink.ModelReference.protect` | `Simulink.ModelReference.modifyProtectedModel`

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.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...  
'sldemo_mdref_counter', 'password');  
Simulink.ModelReference.protect('sldemo_mdref_counter', ...  
'Encrypt', true, 'Report', true);
```

A protected model named `sldemo_mdref_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.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...  
'sldemo_mdref_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.

Version History

Introduced in R2014b

See Also

Simulink.ModelReference.protect |
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForView

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.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.ProtectedModel.setPasswordForView(...  
'sldemo_mdref_counter','password');  
Simulink.ModelReference.protect('sldemo_mdref_counter',...  
'Webview',true,'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdref_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.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.ProtectedModel.setPasswordForView(...  
'sldemo_mdref_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.

Version History

Introduced in R2014b

See Also

Simulink.ModelReference.protect |
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

Simulink.ProtectedModel.addTarget

Add code generation support for current target to protected model

Syntax

```
Simulink.ProtectedModel.addTarget mdl  
Simulink.ProtectedModel.addTarget mdl, Verbose=tf)
```

Description

`Simulink.ProtectedModel.addTarget(mdl)` adds code generation support for the target of the current model, specified by `mdl`, to a protected model of the same name. Each target that the protected model supports is identified by the root of the **System target file** configuration 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**.

`Simulink.ProtectedModel.addTarget(mdl, Verbose=tf)` lets you specify whether to display verbose build process information during model protection.

Examples

Add Target to Protected Model

Add the currently configured model target to the protected model.

Load an example model.

```
mdl = "sldemo_mdref_counter";  
openExample(mdl);
```

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(...  
mdl, "password");
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect(mdl, Mode="CodeGeneration", ...  
Modifiable=true, Report=true);
```

Get a list of targets that the protected model supports.

```
st = Simulink.ProtectedModel.getSupportedTargets mdl)
```

Configure the unprotected model to support a new target.

```
set_param mdl, SystemTargetFile="ert.tlc");
save_system mdl);
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget mdl);
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets mdl)
```

Get Verbose Build Process Information

Add the currently configured model target to the protected model.

Load an example model.

```
mdl = "sldemo_mdlref_counter";
openExample mdl);
```

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(...
mdl, "password");
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect mdl, Mode="CodeGeneration", ...
Modifiable=true, Report=true, Verbose=true);
```

Get a list of targets that the protected model supports.

```
st = Simulink.ProtectedModel.getSupportedTargets mdl)
```

Configure the unprotected model to support a new target.

```
set_param mdl, SystemTargetFile="ert.tlc");
save_system mdl);
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget mdl, Verbose=true);
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets mdl)
```

Input Arguments

mdl — Model name

string or character vector

Model name, specified as a string or character vector that contains the name of a model or the path name of a Model block that references the protected model.

tf — Verbose output of build process information

false or 0 (default) | true or 1

Verbose output of build process information, specified as a numeric or logical 0 (false) or 1 (true).

By default, the model protection process displays limited build process information. To display verbose build process information during model protection, set **Verbose** to **true**.

To control the display of build process information, the model protection process temporarily overrides the values of the **Verbose accelerator builds** and **Verbose build** configuration parameters.

Data Types: `logical`

Version History

Introduced in R2015a

R2024b: Model protection now provides limited build process information by default

Behavior changed in R2024b

To focus on the high-level steps in the model protection process, the `Simulink.ProtectedModel.addTarget` function now displays limited build process information by default. The model protection process temporarily overrides the values of the **Verbose accelerator builds** and **Verbose build** configuration parameters.

To display verbose build process information during model protection, set **Verbose** to **true**.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.removeTarget` |
`Simulink.ProtectedModel.getSupportedTargets` |
`Simulink.ProtectedModel.getCurrentTarget` |
`Simulink.ProtectedModel.setCurrentTarget` | `Simulink.ProtectedModel.getConfigSet`

Topics

“Create Protected Models with Multiple Targets”

Simulink.ProtectedModel.Callback

Callback code that executes in response to protected model events

Description

For a specific 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.

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.

Creation

Syntax

```
Simulink.ProtectedModel.Callback(Event,AppliesTo,CallbackText)
Simulink.ProtectedModel.Callback(Event,AppliesTo,callbackFile)
```

Description

`Simulink.ProtectedModel.Callback(Event,AppliesTo,CallbackText)` creates a callback object for a specific protected model functionality and event. The `CallbackText` specifies MATLAB commands to execute for the callback.

`Simulink.ProtectedModel.Callback(Event,AppliesTo,callbackFile)` creates a callback object for a specific protected model functionality and event. The `CallbackFileName` specifies a MATLAB script to execute for the callback. The script must be on the MATLAB path.

Properties

AppliesTo — Protected model functionality

'AUTO' (default) | 'CODEGEN' | 'SIM' | 'VIEW'

Protected model functionality that the event applies to, specified as one of these 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 for only 'CODEGEN' functionality.

Example: 'SIM'

CallbackFileName — Callback script to execute

character vector | string scalar

MATLAB script to execute in response to an event, specified as a character vector or string scalar. The script must be on the MATLAB path.

Example: 'pmCallback.m'

CallbackText — Callback code to execute

character vector | string scalar

MATLAB commands to execute in response to an event, specified as a character vector or string scalar.

Example: 'A = [15 150];disp(A)'

Event — Event that triggers callback

'PreAccess' | 'Build'

Event that triggers the callback, specified as one of these options:

- 'PreAccess': Callback code is executed before simulation, build, or read-only viewing.
- 'Build': Callback code is executed before build. Valid for only 'CODEGEN' functionality.

Example: 'PreAccess'

OverrideBuild — Option to override protected model build

false (default) | true

Option to override the protected model build process, specified as a Boolean value. The override option applies only to a callback object that you define for a 'Build' event for the 'CODEGEN' functionality. You set this option by using the `setOverrideBuild` method.

Object Functions

`setOverrideBuild` Override protected model build

Examples

Create Protected Model by Using a Callback

- 1 Create the callback object.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess',...  
        'SIM','disp(''Hello world!'')')
```

- 2 Protect the model `sldemo_mdhref_counter` and specify the callback object.

```
openExample('sldemo_mdhref_counter');  
Simulink.ModelReference.protect('sldemo_mdhref_counter',...  
        'Callbacks',{pmCallback})
```

- 3 Simulate the model `sldemo_mdhref_basic`, which references the protected model that you created.

```
openExample('sldemo_mdhref_basic');  
sim('sldemo_mdhref_basic')
```

For each instance of the protected model reference in the top model, the output is displayed.

```
Hello world!
Hello world!
Hello world!
```

Create Protected Model That Uses a Callback Script

- 1 On the MATLAB path, create a callback script `pm_callback.m` that contains this code:

```
disp('Hello world!')
```

- 2 Create a callback object that uses the script for the callback code. Protect the model `sldemo_mdref_counter` and specify the callback object.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdref_counter',...
'Mode','CodeGeneration','Callbacks',{pmCallback})
```

The callback script executes during the code generation phase of the model protection process.

- 3 Generate code for the model `sldemo_mdref_basic`, which references the protected model that you created.

```
slbuild('sldemo_mdref_basic')
```

During the code generation phase for the referenced protected model, code in `pm_callback.m` executes.

Version History

Introduced in R2016a

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getCallbackInfo`

Topics

“Define Callbacks for Protected Models”

“Protect Models to Conceal Contents”

“Code Generation Requirements and Limitations”

setOverrideBuild

Namespace: Simulink.ProtectedModel

Override protected model build

Syntax

```
setOverrideBuild(callback, override)
```

Description

`setOverrideBuild(callback, 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.

Examples

Create Code Generation Callback to Override Build Process

- 1 Create a callback object that uses a character vector of MATLAB commands for the callback code. Define the callback for a 'Build' event for the 'CODEGEN' functionality.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...  
'CODEGEN','disp('Hello world!')')
```

- 2 Specify that the callback override the build process.

```
setOverrideBuild(pmCallback, true);
```

- 3 Protect the model `sldemo_mdref_counter` and specify the callback that you created.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.protect('sldemo_mdref_counter',...  
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

- 4 Build the model `sldemo_mdref_basic`, which references the protected model `sldemo_mdref_counter`. When the top model begins to build the protected model, the callback that you created overrides the build process.

```
openExample('sldemo_mdref_basic');  
slbuild('sldemo_mdref_basic')
```

Input Arguments

callback — Protected model callback

`Simulink.ProtectedModel.Callback` object

Protected model callback that you want to override the protected model build process, specified as a `Simulink.ProtectedModel.Callback`. The callback object must be defined for a 'Build' event for 'CODEGEN' functionality.

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 to only a callback object defined for a 'Build' event for the 'CODEGEN' functionality.

Example: `pmcallback.setOverrideBuild(true)`

Version History

Introduced in R2016a

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.Callback`

Topics

“Define Callbacks for Protected Models”

“Protect Models to Conceal Contents”

“Code Generation Requirements and Limitations”

Simulink.ProtectedModel.CallbackInfo

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.

Creation

Syntax

```
Simulink.ProtectedModel.getCallbackInfo(ModelName,Event,Functionality)
```

Description

`Simulink.ProtectedModel.getCallbackInfo(ModelName,Event,Functionality)` creates a `Simulink.ProtectedModel.CallbackInfo` object for the callback that applies to the protected model for the event and functionality that you specify.

Properties

CodeInterface — Code interface generated by protected model

'Top model' | 'Model reference'

Code interface that the protected model generates, specified as 'Top model' or 'Model reference'.

Event — Event that triggered callback

'PreAccess' | 'Build'

Event that triggered the callback, specified as one of these values:

- 'PreAccess': Callback code executed before simulation, build, or read-only viewing.
- 'Build': Callback code executed before build. This option is valid only for the 'CODEGEN' functionality.

Functionality — Protected model functionality

'AUTO' (default) | 'CODEGEN' | 'SIM' | 'VIEW'

Protected model functionality that the event applies to, specified as one of these 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 for only 'CODEGEN' functionality.

ModelName — Protected model name

character vector | string scalar

Protected model name, specified as a character vector or string scalar.

Example: 'myProtectedModel.slxp'

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 for only code generation callbacks.

Example: 'ert'

Object Functions

getBuildInfoForModel Build information object for specified model

Examples**Use Protected Model Information in Simulation Callback**

- 1 On the MATLAB path, create a callback script `pm_callback.m` that contains this code:

```
s1 = 'Simulating protected model: ';
cbinfoobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdref_counter','PreAccess','SIM');
disp([s1 cbinfoobj.ModelName])
```

The script uses the `Simulink.ProtectedModel.CallbackInfo` object to display the name of the protected model.

- 2 Create a callback object that uses the script for the callback code. Protect the model `sldemo_mdref_counter` and specify the callback object.

```
openExample('sldemo_mdref_counter');
pmCallback = Simulink.ProtectedModel.Callback('PreAccess'...
,'SIM', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdref_counter',...
'Callbacks',{pmCallback})
```

- 3 Simulate the protected model. When each instance of the protected model reference in the top model is simulated, the output from the callback is listed.

```
openExample('sldemo_mdref_basic');
sim('sldemo_mdref_basic')

Simulating protected model: sldemo_mdref_counter
Simulating protected model: sldemo_mdref_counter
Simulating protected model: sldemo_mdref_counter
```

Version History

Introduced in R2016a

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getCallbackInfo`

Topics

"Define Callbacks for Protected Models"

"Protect Models to Conceal Contents"

"Code Generation Requirements and Limitations"

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_mdref_counter','Build','CODEGEN');
disp([s1 cbinfobj.CodeInterface]);
```

When you create a protected model with a simulation callback, use the script.

```
openExample('sldemo_mdref_basic');
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

Build the protected model. Before the start of the protected model build process, the code interface is displayed.

```
slbuild('sldemo_mdref_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

cbinfoobj — Callback information object

`Simulink.ProtectedModel.CallbackInfo`

Callback information, specified as a `Simulink.ProtectedModel.CallbackInfo` object.

Version History

Introduced in R2016a

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

Topics

“Define Callbacks for Protected Models”

“Protect Models to Conceal Contents”

“Code Generation Requirements and Limitations”

getBuildInfoForModel

Namespace: Simulink.ProtectedModel

Build information object for specified model

Syntax

```
bldobj = getBuildInfoForModel(callbackInfo, model)
```

Description

`bldobj = getBuildInfoForModel(callbackInfo, model)` returns the `RTW.BuildInfo` object that specifies the build toolchain and arguments for the model. 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 for only code generation callbacks in response to a 'Build' event.

Examples

Get Build Information from a Code Generation Callback

- 1 On the MATLAB path, create a callback script, `pm_callback.m`, that contains this code:

```
cbinfoobj = Simulink.ProtectedModel.getCallbackInfo(...
    'sldemo_mdref_counter', 'Build', 'CODEGEN');
bldinfo = cbinfoobj.getBuildInfoForModel(cbinfoobj.ModelName);
buildargs = getBuildArgs(bldinfo)
```

- 2 Create a callback object that uses the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
    'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdref_counter',...
    'Mode', 'CodeGeneration', 'Callbacks', {pmCallback})
```

- 3 Build the protected model. Before the start of the protected model build, the build arguments are displayed.

```
openExample('sldemo_mdref_basic');
slbuild('sldemo_mdref_basic')
```

Input Arguments

callbackInfo — Callback information object

`Simulink.ProtectedModel.CallbackInfo` object

Callback information object, specified as a `Simulink.ProtectedModel.CallbackInfo` object. The callback object must be defined for a 'Build' event for the 'CODEGEN' functionality.

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.

Output Arguments

blldobj — Build toolchain and arguments

`RTW.BuildInfo` object

Build toolchain and arguments, returned as a `RTW.BuildInfo` object.

Version History

Introduced in R2016a

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

Topics

“Define Callbacks for Protected Models”

“Protect Models to Conceal Contents”

“Code Generation Requirements and Limitations”

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.

```
openExample('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.

```
openExample('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

Version History

Introduced in R2015a

See Also

Simulink.ModelReference.protect | Simulink.ProtectedModel.addTarget |
Simulink.ProtectedModel.removeTarget |
Simulink.ProtectedModel.getSupportedTargets |
Simulink.ProtectedModel.getCurrentTarget |
Simulink.ProtectedModel.setCurrentTarget

Topics

“Create Protected Models with Multiple Targets”
“Reference Protected Models from Third Parties”

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.

```
openExample('sldemo_mdref_counter');  
save_system('sldemo_mdref_counter','mdref_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(...  
'mdref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdref_counter','Mode',...  
'CodeGeneration','Modifiable',true,'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdref_counter','SystemTargetFile','ert.tlc');  
save_system('mdref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdref_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.

Version History

Introduced in R2015a

See Also

Simulink.ModelReference.protect | Simulink.ProtectedModel.addTarget |
Simulink.ProtectedModel.removeTarget |
Simulink.ProtectedModel.getSupportedTargets |
Simulink.ProtectedModel.setCurrentTarget | Simulink.ProtectedModel.getConfigSet

Topics

“Create Protected Models with Multiple Targets”
“Reference Protected Models from Third Parties”

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.

```
openExample('sldemo_mdref_counter');  
save_system('sldemo_mdref_counter','mdref_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(...  
'mdref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdref_counter','Mode',...  
'CodeGeneration','Modifiable',true,'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdref_counter','SystemTargetFile','ert.tlc');  
save_system('mdref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdref_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.

Version History

Introduced in R2015a

See Also

Simulink.ModelReference.protect | Simulink.ProtectedModel.addTarget |
Simulink.ProtectedModel.removeTarget | Simulink.ProtectedModel.getCurrentTarget
| Simulink.ProtectedModel.setCurrentTarget |
Simulink.ProtectedModel.getConfigSet

Topics

“Create Protected Models with Multiple Targets”

“Reference Protected Models from Third Parties”

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.

```
openExample('sldemo_mdhref_counter');  
save_system('sldemo_mdhref_counter','mdhref_counter.slx');
```

Create a protected model enabling support for code generation and reporting.

```
Simulink.ModelReference.protect('mdhref_counter','Mode',...  
'CodeGeneration','Report',true);
```

Open the protected model without specifying how to view it.

```
Simulink.ProtectedModel.open('mdhref_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.

```
openExample('sldemo_mdref_counter');  
save_system('sldemo_mdref_counter','mdref_counter.slx');
```

Create a protected model with support for code generation, Web view, and reporting.

```
Simulink.ModelReference.protect('mdref_counter','Mode',...  
'CodeGeneration','Webview',true,'Report',true);
```

Open the protected model and specify that you want to see the Web view.

```
Simulink.ProtectedModel.open('mdref_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.

Version History

Introduced in R2015a

See Also

Simulink.ModelReference.protect

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.

```
openExample('sldemo_mdref_counter');  
save_system('sldemo_mdref_counter','mdref_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(...  
'mdref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdref_counter','Mode',...  
'CodeGeneration','Modifiable',true,'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdref_counter','SystemTargetFile','ert.tlc');  
save_system('mdref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdref_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.

Version History

Introduced in R2015a

See Also

Simulink.ModelReference.protect | Simulink.ModelReference.modifyProtectedModel |
Simulink.ProtectedModel.addTarget | Simulink.ProtectedModel.getSupportedTargets
| Simulink.ProtectedModel.getCurrentTarget |
Simulink.ProtectedModel.setCurrentTarget | Simulink.ProtectedModel.getConfigSet

Topics

“Create Protected Models with Multiple Targets”

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.

```
openExample('sldemo_mdref_counter');  
save_system('sldemo_mdref_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.

Version History

Introduced in R2015a

See Also

Simulink.ModelReference.protect | Simulink.ProtectedModel.addTarget |
Simulink.ProtectedModel.removeTarget |
Simulink.ProtectedModel.getSupportedTargets |
Simulink.ProtectedModel.getCurrentTarget | Simulink.ProtectedModel.getConfigSet

Topics

“Create Protected Models with Multiple Targets”
“Reference Protected Models from Third Parties”

Simulink.ProtectedModel.sign

Attach digital signature to protected model

Syntax

```
Simulink.ProtectedModel.sign(protectedModel,certificateFile)
```

Description

`Simulink.ProtectedModel.sign(protectedModel,certificateFile)` attaches a digital signature with the certificate `certificateFile` to the protected model `protectedModel`.

Examples

Sign a Protected Model

Protect a model, and then digitally sign it with a certificate.

Open and protect the model that you want to sign. For this example, protect the model `sldemo_mdref_counter`.

```
openExample('sldemo_mdref_counter');  
Simulink.ModelReference.protect('sldemo_mdref_counter');
```

Locate the certificate file that you want to use to sign the protected model. Sign the model by using the certificate file.

```
Simulink.ProtectedModel.sign('sldemo_mdref_counter.slxp','certificate_file.pfx');
```

In the dialog box, enter the password for the certificate file.

Input Arguments

protectedModel — Name of protected model

character vector | string scalar

Name of the protected model that you want to sign, specified as a character vector or string scalar. The protected model has a `.slxp` extension.

Example: `'my_model.slxp'`

certificateFile — Certificate file to use for signing

character vector | string scalar

Certificate file to use for signing the protected model, specified as a character vector or string scalar. The certificate must be a PKCS #12 file with the extension `.pfx` or `.p12`.

Example: `'my_cert.pfx'`

Example: `'InstitutionCertificate.p12'`

Version History

Introduced in R2020a

See Also

`Simulink.ModelReference.protect`

Topics

"Sign a Protected Model"

slConfigUIGetVal

Return current value for any model configuration parameter

Syntax

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

Description

`value = slConfigUIGetVal(hDlg,hSrc,'OptionName')` returns the value currently set in the dialog for any model configuration parameter.

The `slConfigUIGetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when you:

- Change System Target Files.
- Build the model.

Examples

Get Configuration Option Value

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);
    hSrc.getConfigSet.refreshDialog;
```

Input Arguments

hDlg — handle for STF callback functions

handle

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.

Example: `hDlg`

hSrc — handle for STF callback functions

handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable and use it to refresh the Configuration Parameters dialog box. Do not set it or use it for another purpose.

Example: `hSrc`

OptionName — TLC variable

TLC variable

Quoted name of the TLC variable defined for a custom target configuration option.

Example: `'myTLCvariable'`

Output Arguments

value — Option value

option value

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

Version History

Introduced in R2006b

See Also

`slConfigUISetEnabled` | `slConfigUISetVal`

Topics

“Define and Display Custom Target Options”

“Custom Target Optional Features”

slConfigUISetEnabled

Enable or disable any model configuration parameter

Syntax

```
slConfigUISetEnabled(hDlg,hSrc,'OptionName',value)
```

Description

`slConfigUISetEnabled(hDlg,hSrc,'OptionName',value)` 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 model configuration parameter. To refresh the Configuration Parameters dialog, use `hSrc.getConfigSet.refreshDialog`.

Examples

Disable Model Configuration Option

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'), ...
        ' Clear and disable "Terminate function required"']);

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

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

Input Arguments

hDlg — Handle for STF callback

handle

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.

Example: `hDlg`

hSrc — Handle for STF callback

handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable and use it to refresh the Configuration Parameters dialog box. Do not set it or use it for another purpose.

Example: hSrc

'OptionName', value — TLC variable name and set enable
false | true

Quoted name and set enable of the TLC variable defined for a model configuration parameter.

Example: 'myConfigVariable', false

Version History

Introduced in R2006b

See Also

slConfigUIGetVal | slConfigUISetVal

Topics

“Define and Display Custom Target Options”

“Custom Target Optional Features”

slConfigUISetVal

Set value for any model configuration parameter

Syntax

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

Description

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

Set Configuration Option Value

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'), ...
        ' Clear and disable "Terminate function required".']);

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

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

Input Arguments

hDlg — Handle for STF callback

handle

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.

Example: `hDlg`

hSrc — Handle for STF callback

handle

Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable and use it to refresh the Configuration Parameters dialog box. Do not set it or use it for another purpose.

Example: hSrc

'OptionName', value — TLC variable name and value

option value

Quoted name and value of the TLC variable defined for a model configuration parameter.

Example: 'myConfigVariable',1

Version History

Introduced in R2006b

See Also

slConfigUIGetVal | slConfigUISetEnabled

Topics

“Define and Display Custom Target Options”

“Custom Target Optional Features”

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 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 default ERT target
model='CounterModel';
openExample(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 `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 = 'myTMF';`

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';`

Version History

Introduced in R2009b

See Also

`getActiveConfigSet` | `ConfigSet` | `Simulink.ConfigSetRef`

Topics

“Select a System Target File Programmatically”

“Configure a System Target File”

“Set Target Language Compiler Options”

target Namespace

Manage target hardware and build tool information

Description

Use these classes to manage target hardware and build tool information. For example, register new target hardware for code generation, set up target connectivity for external mode and processor-in-the-loop (PIL) simulations, or create custom CMake toolchain definitions for building generated code on your development computer.

Classes

target.AddOn	Describe add-on properties for target type
target.Alias	Create alternative identifier for target object
target.API	Describe API details
target.APIImplementation	Describe API implementation details
target.ApplicationExecutionTool	Capture system command information to run application from MATLAB computer
target.ApplicationStatus	Describe status of application on target hardware
target.Board	Provide hardware board details
target.Breakpoint	Provide breakpoint details for debugger
target.BuildDependencies	Describe C and C++ build dependencies to associate with target hardware
target.BuildTool	Describe build tool
target.BuildToolType	Describe build tool type
target.CMake	Specify CMake installation for building generated code
target.CMakeBuildType	Describe CMake build type or build configuration
target.CMakeBuilder	Configure how CMake builds generated code
target.CMakeCacheEntry	Configure a CMake cache entry
target.Command	Capture system command for execution on MATLAB computer
target.CommunicationChannel	Describe communication channel properties
target.CommunicationInterface	Describe data I/O details for target hardware
target.CommunicationProtocolStack	Describe communication protocol parameters
target.Connection	Base class for target connection properties
target.ConnectionProperties	Describe target-specific connection properties
target.DebugExecutionTool	Provide MATLAB service interface for debugger to manage processes on target hardware
target.DebugIOTool	Debug byte stream I/O tool service interface
target.Directive	Describe command-line flag for tool
target.Emulator	Provide target emulator details
target.EnvironmentConfiguration	Configure system environment for toolchain
target.ExecutionService	Describe implementation of execution service for target application
target.ExecutionTool	MATLAB service interface for tool that manages application execution on target hardware
target.ExternalMode	Represent external mode protocol stack
target.ExternalModeConnectivity	Base class for external mode connectivity options

<code>target.FileType</code>	Define identifier for file type
<code>target.FloatingPointDataType</code>	Describe floating point data type implemented by compiler for target hardware
<code>target.Function</code>	Provide function signature information
<code>target.HardwareComponentSupport</code>	Describe support for a hardware component
<code>target.HostProcessExecutionTool</code>	Capture system command information to run target application from MATLAB computer
<code>target.LanguageImplementation</code>	Provide C and C++ compiler implementation details
<code>target.MainFunction</code>	Provide C and C++ dependencies for main function of target hardware application
<code>target.MakefileBuilder</code>	Specify that toolchain is makefile-based
<code>target.MakeToolType</code>	Describe syntax for makefile type
<code>target.MATLABDependencies</code>	Describe MATLAB class and function dependencies
<code>target.PairedDirective</code>	Describe pair of command-line flags
<code>target.Object</code>	Base class for target types
<code>target.OperatingSystem</code>	Describe operating system for target hardware
<code>target.PILProtocol</code>	Describe PIL protocol implementation for target hardware
<code>target.Port</code>	Describe connection via target hardware port
<code>target.PortConnection</code>	Describe target connection port
<code>target.Processor</code>	Provide target processor information
<code>target.ProfilingFreezingOverhead</code>	Capture freezing and unfreezing instrumentation overhead
<code>target.ProfilingFunctionOverhead</code>	Capture function instrumentation overhead
<code>target.ProfilingTaskOverhead</code>	Capture task instrumentation overhead
<code>target.RepeatingDirective</code>	Describe repeated command-line flag for tools
<code>target.RS232Channel</code>	Describe serial communication channel
<code>target.Support</code>	Specify association between target namespace objects
<code>target.SystemCommandExecutionTool</code>	Capture system command information to run target application from MATLAB computer
<code>target.TargetConnection</code>	Provide details about connecting MATLAB computer to target hardware
<code>target.TCPChannel</code>	Describe TCP communication properties
<code>target.Timer</code>	Provide timer details for processor
<code>target.Tools</code>	Describe properties of tools for target hardware
<code>target.Toolchain</code>	Capture high-level information about toolchain
<code>target.UDPChannel</code>	Describe UDP communication
<code>target.XCP</code>	Describe XCP protocol stack for target hardware
<code>target.XCPExternalModeConnectivity</code>	Represent connectivity options in external mode protocol stack
<code>target.XCPPlatformAbstraction</code>	Specify XCP platform abstraction layer for target hardware
<code>target.XCPTCPIPTransport</code>	Represent XCP TCP/IP transport protocol layer
<code>target.XCPTransport</code>	Base class for XCP transport protocol layer
<code>target.XCPSerialTransport</code>	Represent XCP serial transport protocol layer

Functions

<code>target.add</code>	Add target object to internal database
<code>target.clear</code>	Clear all target objects from internal database
<code>target.create</code>	Create target object
<code>target.export</code>	Export target object data
<code>target.get</code>	Retrieve target objects from internal database
<code>target.remove</code>	Remove target object from internal database
<code>target.update</code>	Update target objects in internal database

target.upgrade Upgrade existing definitions of hardware devices

Version History

Introduced in R2019a

See Also

Topics

“Register New Hardware Devices”

“Customize Connectivity for XCP External Mode Simulations”

“Set Up PIL Connectivity by Using Target Framework”

“Create Custom CMake Toolchain Definition”

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.add

Namespace: target

Add target object to internal database

Syntax

```
objectsAdded = target.add(targetObject)
objectsAdded = target.add(targetObject, Name,Value)
```

Description

`objectsAdded = target.add(targetObject)` adds the specified target object to an internal database and returns a vector that contains the added objects. By default, the target data is available only for the current MATLAB session.

`objectsAdded = target.add(targetObject, Name,Value)` uses name-value arguments to control persistence over MATLAB sessions and command-line output.

Examples

Specify Hardware Implementation

To specify a hardware implementation that persists over MATLAB sessions, use the `target.create` and `target.add` functions.

```
myLangImp = target.create('LanguageImplementation', ...
                          'Name', 'MyLanguageImplementation', ...
                          'Copy', 'ARM Compatible-ARM Cortex');

myProc = target.create('Processor','Name','MyProcessor');
myProc.LanguageImplementations = myLangImp;
objectsAdded = target.add(myProc, ...
                          'UserInstall',true, ...
                          'SuppressOutput',true);
```

Input Arguments

targetObject — Target object

object

Specify the target object that you want to add to the internal database.

Example: `target.add(myTargetObject);`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `target.add(myTargetObject, 'UserInstall', true);`

UserInstall — Target data persistence

false (default) | true

Control persistence of target data in an internal database:

- `true` -- Target data persists in internal database over multiple MATLAB sessions.
- `false` -- Target data is in internal database only for the current MATLAB session.

Example: `target.add(myTargetObject, 'UserInstall', true);`

Data Types: logical

SuppressOutput — Control command-line output

false (default) | true

Control command-line output of function:

- `true` -- Suppress command-line output from the function.
- `false` -- Provide information about the objects that the function adds to internal database.

Example: `target.add(myTargetObject, 'SuppressOutput', true);`

Data Types: logical

Output Arguments

objectsAdded — Objects added

object vector

Target objects that the function adds to internal database.

Version History

Introduced in R2019a

See Also

`target.clear` | `target.create` | `target.get` | `target.remove` | `target.update`

Topics

“Register New Hardware Devices”

target.AddOn Class

Namespace: target

Describe add-on properties for target type

Description

Use the `target.AddOn` class to capture custom properties that you can associate with these types of objects:

- `target.CommunicationChannel`
- `target.CommunicationProtocolStack`
- `target.Board`
- `target.Processor`
- `target.ConnectionProperties`

To extend the objects, assign the `target.AddOn` object to the `AddOns` property.

To create a `target.AddOn` object, use the `target.create` function.

Properties

Name — Add-on object name

character vector | string

Name of the reusable add-on object.

Example: `arduinoAddOn.Name = 'ArduinoBoardProperties';`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Methods

Public Methods

`addProperty` Add a custom property to `target.AddOn` object

Examples

Part Number and Programmer for Arduino Board Definition

Add device-specific properties to a `target.Board` definition. Add information about the Arduino® part number and programmer to an Arduino board definition.

Create a board for Arduino Mega 2560.

```
mega = target.create('Board', ...
    'Manufacturer', 'Arduino', ...
    'Name', 'Mega 2560');
```

Create a `target.AddOn` object that specifies the Arduino board part number and programmer.

```
arduinoAddOn = target.create('AddOn');
arduinoAddOn.Name = 'ArduinoBoardProperties';
arduinoAddOn.addProperty('ArduinoPartNumber', 'String');
arduinoAddOn.addProperty('ArduinoProgrammer', 'String');
mega.AddOns = arduinoAddOn;
```

Specify the part number and programmer values.

```
mega.set('ArduinoPartNumber', 'm2560');
mega.set('ArduinoProgrammer', 'wiring');
```

You can parameterize the `avrdude` command for the deployment of an Arduino application.

```
command= target.create('Command');
command.String = 'avrdude';

command.Arguments = {'-p$(BOARD.ArduinoPartNumber)' ...
    '-c$(BOARD.ArduinoProgrammer)' ...
    '-Uflash:w:$(EXE):i'};

mega.Tools.ExecutionTools(1) = target.create('SystemCommandExecutionTool', ...
    'Name', 'avrdude execution', ...
    'StartCommand', command);
```

Version History

Introduced in R2020b

See Also

`target.create`

addProperty

Class: target.AddOn

Namespace: target

Add a custom property to target.AddOn object

Syntax

```
myAddOn.addProperty(propertyName, propertyType)
```

Description

`myAddOn.addProperty(propertyName, propertyType)` adds the property `propertyName` of type `propertyType` to the object *myAddOn*.

Input Arguments

propertyName — Property name

character vector | string

Name of property that you want to add to object.

Example: `exampleAddOn.addProperty('myPrptyName', 'myPrptyType')`

propertyType — Property type

double | string | object

Type of property that you want to add to object.

Example: `exampleAddOn.addProperty('myPrptyName', 'myPrptyType')`

Examples

Add Property to target.AddOn Object

For workflow examples that use this method, see “Examples” on page 2-0 .

Version History

Introduced in R2020b

See Also

target.AddOn

target.Alias Class

Namespace: target

Create alternative identifier for target object

Description

Use the `target.Alias` class to create alternative identifiers for target objects. For example, if a target object has a long class identifier, you can create a `target.Alias` object that provides a short identifier for the target object.

To create a `target.Alias` object, use the `target.create` function.

Properties

Name — Alternative identifier

character vector | string

Alternative identifier for target object.

Attributes:

GetAccess	public
SetAccess	public

For — Target object

object

Original target object.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create Alternative Identifier for Target Object

To create alternative identifiers for target objects, use the `target.Alias` class.

For example, if a `target.Processor` object has a long class identifier, you can create a `target.Alias` object that provides a short identifier for the `target.Processor` object.

Retrieve the `target.Processor` object.

```
processorObj = target.get("Processor", ...
    "Analog Devices-ADSP-CM40x (ARM Cortex-M)");
```

Use the `target.create` function to create a `target.Alias` object.

```
aliasProcessorObj = target.create("Alias");
```

Use `target.Alias` object properties to specify the alternative identifier and original target object.

```
aliasProcessorObj.Name = "myShortName";  
aliasProcessorObj.For = procesorObj;
```

Add the `target.Alias` object to an internal database.

```
objectsAdded = target.add(aliasProcessorObj);
```

```
target.add summary:
```

```
Objects added to internal database for current MATLAB session:  
target.Alias      "myShortName"  
1 object not added because they already exist.
```

To retrieve the original `target.Processor` object, enter:

```
target.get("Processor", "myShortName");
```

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded)
```

```
target.remove summary:
```

```
Objects removed from internal database:  
target.Alias      "myShortName"
```

Version History

Introduced in R2019a

See Also

`target.create`

Topics

“Register New Hardware Devices”

target.API Class

Namespace: target

Describe API details

Description

An API defines a set of entry-point functions for interaction with a software application or service. Use a `target.API` object to provide API details for target definition. Use this class with `target.APIImplementation` to describe how an API is used and built on target hardware.

To create a `target.API` object, use the `target.create` function.

Properties

Name — API name

character vector | string

Name of the API.

Example: `timerApi.Name = 'Linux Timer API';`

Attributes:

GetAccess	public
SetAccess	public

Language — API language

`target.Language` object

Programming language of the API implementation.

Attributes:

GetAccess	public
SetAccess	public

Functions — API Functions

`target.Function` object vector

Vector of `target.Function` objects that describe the set of entry-point functions that make up the API.

Attributes:

GetAccess	public
SetAccess	public

Examples

Describe rtiostream C API

This example provides implementation details for the `rtiostream` C API.

```
apiImp = target.create('APIImplementation', 'Name', ...  
                      'x86_rtiostream Implementation');  
apiImp.API = target.create('API', 'Name', 'rtiostream');  
apiImp.BuildDependencies = target.create('BuildDependencies');  
apiImp.BuildDependencies.SourceFiles = ...  
    {fullfile('${MATLAB_ROOT}', 'toolbox', ...  
              'coder', 'rtiostream', 'src', ...  
              'rtiostreamtcpip', 'rtiostream_tcpip.c')};  
apiImp.MainFunction = target.create('MainFunction', ...  
                                   'Name', 'TCP RtIOStream Main');  
apiImp.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
```

Version History

Introduced in R2020b

See Also

[target.create](#) | [target.APIImplementation](#) | [target.Function](#) |
[target.LanguageImplementation](#)

target.APIImplementation Class

Namespace: target

Describe API implementation details

Description

An API defines a set of entry-point functions for interaction with a software application or service. Use a `target.APIImplementation` object to provide details about how an API is used and built on target hardware.

To create a `target.APIImplementation` object, use the `target.create` function.

Properties

Name — API implementation name

character vector | string

Name of the `APIImplementation` object, which `target.get` uses as an identifier in data retrieval.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

API — API description

`target.API` object

Description of the API implementation definition.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

BuildDependencies — API implementation dependencies

`target.Dependencies` object

Source files, header files, and other dependencies that are needed for building and running the API on the target hardware.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

MainFunction — main function requirements for API implementation

`target.MainFunction` object

Capture run-time dependencies such as main function arguments, initialization code, and main function build dependencies.

Attributes:

GetAccess	public
SetAccess	public

Examples**Describe rtiostream C API**

This example provides implementation details for the `rtiostream` C API.

```
apiImp = target.create('APIImplementation', 'Name', ...  
                      'x86 rtiostream Implementation');  
apiImp.API = target.create('API', 'Name', 'rtiostream');  
apiImp.BuildDependencies = target.create('BuildDependencies');  
apiImp.BuildDependencies.SourceFiles = ...  
    {fullfile('${MATLAB_ROOT}', 'toolbox', ...  
              'coder', 'rtiostream', 'src', ...  
              'rtiostreamtcpip', 'rtiostream_tcpip.c')};  
apiImp.MainFunction = target.create('MainFunction', ...  
                                   'Name', 'TCP RtIOStream Main');  
apiImp.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
```

Version History

Introduced in R2020b

See Also

`target.create` | `target.API` | `target.BuildDependencies` | `target.MainFunction`

target.ApplicationExecutionTool Class

Namespace: target

Capture system command information to run application from MATLAB computer

Description

Use the target.ApplicationExecutionTool class to capture system command information that is required to run an application from your development computer.

Class Attributes

Abstract	true
HandleCompatible	true

For information on class attributes, see “Class Attributes”.

Properties

Name — Execution tool name

character vector | string

Name of the execution tool.

Attributes:

GetAccess	public
SetAccess	public

Id — Object identifier

character vector | string

Value of the Name property.

Attributes:

GetAccess	public
SetAccess	private

Examples

Use target.ApplicationExecutionTool to Describe Application Execution

For an example that uses the target.ApplicationExecutionTool class to describe the application execution on the target hardware, see “Use Debugger for PIL Target Connectivity”. This

workflow uses `target.ExecutionService`, which inherits from `target.ApplicationExecutionTool`.

Version History

Introduced in R2020b

See Also

`target.create` | `target.Command` | `target.HostProcessExecutionTool` | `target.SystemCommandExecutionTool`

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.ApplicationStatus Class

Namespace: target

Describe status of application on target hardware

Description

Use the `target.ApplicationStatus` enumeration class to describe the status of your target application. The enumeration class contains these members.

Member	Application Status
Running	Application running on target hardware
Stopped	Application not running on target hardware
Unknown	Not known

Creation

`target.ApplicationStatus.MemberName` creates an object of the enumeration class.

Examples

PIL Target Connectivity with Debugger

For an example that uses the `target.ApplicationStatus` class, see “Use Debugger for PIL Target Connectivity”.

Version History

Introduced in R2021a

See Also

`target.ExecutionTool`

Topics

“Define Enumeration Classes”

target.Board Class

Namespace: target

Provide hardware board details

Description

Use a `target.Board` object to provide MATLAB with data about your target hardware board, for example, CPU, communication, and tool information.

To create a `target.Board` object, use the `target.create` function.

Properties

Name — Board name

character vector | string

Name of the `target.Board` object, which `target.get` uses as an identifier in data retrieval.

Attributes:

GetAccess	public
SetAccess	public

Processors — Available processors

`target.Processor` object array

Array of `target.Processor` objects that provide descriptions of available processors for the board.

Attributes:

GetAccess	public
SetAccess	public

CommunicationInterfaces — Available communication interfaces

`target.CommunicationInterfaces` object array

Array of `target.CommunicationInterface` objects that provide descriptions of available communication interfaces for the board.

Attributes:

GetAccess	public
SetAccess	public

CommunicationProtocolStacks — Available communication protocols supported by board

`target.CommunicationProtocolStack` object array

Array of `target.CommunicationProtocolStack` objects that provide descriptions of the communication protocols for the board.

Attributes:

GetAccess	public
SetAccess	public

MainFunctions — main function requirements

target.MainFunction object array

Array of target.MainFunction objects that capture run-time dependencies such as main function arguments, initialization code, and main function build dependencies.

Attributes:

GetAccess	public
SetAccess	public

Tools — Tooling for interaction with board

target.Tools object

Collection of tooling descriptions associated with the board. For example, ApplicationExecutionTool to enable execution of applications on the target hardware.

Attributes:

GetAccess	public
SetAccess	public

Examples**Create Board Description**

Create a description of a target hardware board. This code from “Set Up PIL Connectivity by Using Target Framework” shows how to create the description.

Create a board object that provides MATLAB with a description of processor attributes.

```
hostTarget = target.create('Board', 'Name', 'Host Intel processor');
```

Specify the processor for the board, for example, by reusing a supported processor.

```
hostTarget.Processors = target.get('Processor', ...
    'Intel-x86-64 (Linux 64)');
```

Create Communication Interface for Target Hardware

Create a communication interface for the target hardware board. This code snippet from “Set Up PIL Connectivity by Using Target Framework” shows how to create the interface.

```
comms = target.create('CommunicationInterface');

comms.Name = 'Linux TCP Interface';
comms.Channel = 'TCPChannel';
comms.APIImplementations = target.create('APIImplementation', ...
    'Name', 'x86 RTIOStream Implementation');
comms.APIImplementations.API = target.create('API', 'Name', 'RTIO Stream');

...
```

```
hostTarget.CommunicationInterfaces = comms;
```

Specify PIL Protocol Information

Specify PIL protocol information. This code snippet from “Set Up PIL Connectivity by Using Target Framework” shows how to specify the information.

```
pilProtocol = target.create('PILProtocol');  
pilProtocol.Name = 'Linux PIL Protocol';  
pilProtocol.SendBufferSize = 50000;  
pilProtocol.ReceiveBufferSize = 50000;  
hostTarget.CommunicationProtocolStacks = pilProtocol;
```

Version History

Introduced in R2020b

See Also

`target.create` | `target.Processor`

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.Breakpoint Class

Namespace: target

Provide breakpoint details for debugger

Description

Use the target.Breakpoint class to provide breakpoint details for a debugger.

To create a target.Breakpoint object, use the target.create function.

Properties

Function — Breakpoint function

string

Function that contains the breakpoint.

Attributes:

GetAccess	public
SetAccess	public

File — Breakpoint file

string

Path to the file that contains the breakpoint.

Attributes:

GetAccess	public
SetAccess	public

Line — Breakpoint line number

scalar

Line number of the breakpoint.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int32

Examples

PIL Target Connectivity with Debugger

For an example that uses the target.Breakpoint class, see “Use Debugger for PIL Target Connectivity”.

Version History

Introduced in R2021a

See Also

`target.create` | `target.DebugIOTool`

Topics

“Use Debugger for PIL Target Connectivity”

target.BuildDependencies Class

Namespace: target

Describe C and C++ build dependencies to associate with target hardware

Description

Use the `target.BuildDependencies` object to:

- Describe C and C++ build dependencies, for example, source files and include paths.
- Associate the dependencies with your target hardware.

For example, you can use a `target.BuildDependencies` object to describe the build dependencies for a `target.APIImplementation` object.

Properties

CompilerFlags — Compiler flag dependencies

cell array of character vectors | string array

Compiler flag dependencies.

Attributes:

GetAccess	public
SetAccess	public

Defines — Macro definition dependencies

cell array of character vectors | string array

Macro definition dependencies.

Attributes:

GetAccess	public
SetAccess	public

IncludeFiles — Include file dependencies

cell array of character vectors | string array

Include file dependencies.

Attributes:

GetAccess	public
SetAccess	public

IncludePaths — Header file include path dependencies

cell array of character vectors | string array

Header file include path dependencies.

Attributes:

GetAccess	public
SetAccess	public

LinkerFlags — Linker flag dependencies

cell array of character vectors | string array

Linker flag dependencies.

Attributes:

GetAccess	public
SetAccess	public

SharedLibraries — Shared library dependencies

cell array of character vectors | string array

Shared library dependencies.

Attributes:

GetAccess	public
SetAccess	public

SourceFiles — Source file dependencies

cell array of character vectors | string array

Specify path to source files.

Attributes:

GetAccess	public
SetAccess	public

StaticLibraries — Static library dependencies

cell array of character vectors | string array

Static library dependencies.

Attributes:

GetAccess	public
SetAccess	public

Examples

Describe Implementation Build Dependencies for rtiostream C API

This example shows how you can describe the implementation build dependencies for the rtiostream C API.

```
apiImp = target.create('APIImplementation', 'Name', ...  
                      'x86 rtiostream Implementation');  
apiImp.API = target.create('API', 'Name', 'rtiostream');  
apiImp.BuildDependencies = target.create('BuildDependencies');
```

```
apiImp.BuildDependencies.SourceFiles = ...  
    {fullfile('${MATLAB_ROOT}', 'toolbox', ...  
        'coder', 'rtiostream','src', ...  
        'rtiostreamtcpip', 'rtiostream_tcpip.c')};  
apiImp.MainFunction = target.create('MainFunction', ...  
    'Name', 'TCP RtIOStream Main');  
apiImp.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
```

Version History

Introduced in R2020b

See Also

[target.create](#) | [target.APIImplementation](#) | [target.APIImplementation](#)

target.BuildTool Class

Namespace: target

Describe build tool

Description

Use a `target.BuildTool` object to describe a build tool. For example, a compiler, linker, or archiver.

To create a `target.BuildTool` object, use the `target.create` function. Create the object in a single step.

```
buildToolObject = target.create('BuildTool', ...  
                                nameOfBuildToolType, ...  
                                buildToolCommandString, ...  
                                propertyXName, propertyXValue, ...  
                                propertyYName, propertyYValue, ...  
                                propertyZName, propertyZValue)
```

nameOfBuildToolType is the name of the `target.BuildToolType` object that you want to specify for the `BuildToolType` property of *buildToolObject*. For example 'C Compiler'.

buildToolCommandString is the build tool command, which includes required arguments and flags. For example 'gcc'.

Properties

Id — Object identifier

string

Unique ID of object in internal database.

Attributes:

GetAccess	public
SetAccess	private

Name — Build tool name

string

Name of build tool.

Attributes:

GetAccess	public
SetAccess	public

SupportedHardware — Target hardware

`target.HardwareComponentSupport` object array

Target hardware for the build tool.

Attributes:

GetAccess	public
SetAccess	public

HostOperatingSystemSupport — Operating systems support

target.HostOperatingSystemSupport object

Development computer operating systems supported by the build tool.

Attributes:

GetAccess	public
SetAccess	public

Command — Build tool command

target.Command object

Command that invokes the build tool, which includes standard arguments that the command uses.

Attributes:

GetAccess	public
SetAccess	public

BuildToolType — Build tool type

target.BuildToolType

Type or role of the tool in the toolchain. For example, C Compiler, Assembler, or Linker.

Attributes:

GetAccess	public
SetAccess	public

FileExtensions — File extensions

target.FileExtensionSet object array

The file extensions for each type of input and output file that the tool processes.

Attributes:

GetAccess	public
SetAccess	public

Directives — Tool directives

target.BaseDirective object array

Directives (command-line flags) for the tool.

Attributes:

GetAccess	public
SetAccess	public

FileSep — File separator

string

Preferred file separator for the tool.

Attributes:

GetAccess	public
SetAccess	public

Methods**Public Methods**

getDirective	Search for directive
getFileExtensionSet	Search for file type
setDirective	Set directive values
setFileExtensions	Set file extensions
showDirectives	Print directives to screen
showFileExtensions	Print file extensions to screen

Examples**Use target.BuildTool to Specify C Compiler**

This code snippet shows how you can use a `target.BuildTool` object to specify a C compiler that uses command files and generates object files with the `.obj` extension.

```
cCompiler = target.create('BuildTool', 'C Compiler', 'gcc', ...  
    'Name', 'MinGW GCC C Compiler');  
cCompiler.setDirective('CommandFile', '@');  
cCompiler.setFileExtensions('Object', {' .obj' });
```

Version History

Introduced in R2023a

See Also

`target.BuildToolType` | `target.Directive` | `target.FileType` | `target.MakefileBuilder` | `target.MakeToolType` | `target.PairedDirective` | `target.RepeatingDirective` | `target.Toolchain`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

getDirective

Class: target.BuildTool

Namespace: target

Search for directive

Syntax

```
directiveObject = myBuildTool.getDirective(directiveName)
```

Description

`directiveObject = myBuildTool.getDirective(directiveName)` searches for the specified directive in the Directives property of the `target.BuildTool` object and returns a `target.BaseDirective` object.

Input Arguments

directiveName — Directive name

character vector | string

Name of directive.

Example: `objectName = cCompiler.getDirective('CommandFile');`

Output Arguments

directiveObject — Directive object

`target.BaseDirective` object

`target.BaseDirective` object that contains the directive.

Version History

Introduced in R2023a

See Also

`setDirective` | `showDirectives`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

getFileExtensionSet

Class: target.BuildTool

Namespace: target

Search for file type

Syntax

```
fileExtensionsObject = myBuildTool.getFileExtensionSet(fileName)
```

Description

`fileExtensionsObject = myBuildTool.getFileExtensionSet(fileName)` searches for the named file type in the `FileExtensions` property of the `target.BuildTool` object and returns a `target.FileExtensionSet` object.

Input Arguments

fileName — File type

character vector | string

Name of file type.

Example: `fileExtsObj = cCompiler.getFileExtensionSet('Object');`

Output Arguments

fileExtensionsObject — File extensions object

`target.FileExtensionSet` object

`target.FileExtensionSet` object that contains the named file type.

Version History

Introduced in R2023a

See Also

`setFileExtensions` | `showFileExtensions`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

setDirective

Class: target.BuildTool

Namespace: target

Set directive values

Syntax

```
myBuildTool.setDirective(directiveName,directiveValue)
myBuildTool.setDirective(pairedDirectiveName,startValue,endValue)
```

Description

`myBuildTool.setDirective(directiveName,directiveValue)` sets the named directive to the specified value.

`myBuildTool.setDirective(pairedDirectiveName,startValue,endValue)` sets the start and end values of the named directive.

If the `Directives` property does not contain the directive, the method appends a new object of the appropriate subclass of `target.BaseDirective`.

If the directive name is not supported by the build tool type specified by the `target.BuildToolType` object, the method produces an error.

Input Arguments

directiveName — Directive name

character vector | string

Name of directive. This table provides a list of supported directives.

Build Tool Type	Directive Name (Command-Line Flag)	Optional or Required	Directive Purpose	Example Value
'Assembler'	'Debug'	Required	Generate debugging information.	'-g'
	'IncludeSearchPath'	Required	Prefix header file include path specification.	'-I'
	'OutputFlag'	Required	Prefix name of tool output (object file).	'-o'
	'PreprocessorDefine'	Required	Prefix preprocessor macro definitions.	'-D'
	'GenerateDependencyFiles'	Optional	Generate file that describes header file dependencies.	'-MMD'

Build Tool Type	Directive Name (Command-Line Flag)	Optional or Required	Directive Purpose	Example Value
'C Compiler' or 'C++ Compiler'	'CompileFlag'	Required	Compile source file to produce object file.	'-c'
	'Debug'	Required	Generate debugging information.	'-g'
	'DisableOptimization'	Required	Disable compiler optimizations.	'-O0'
	'EnableOptimization'	Required	Enable compiler optimizations.	'-O3'
	'IncludeSearchPath'	Required	Prefix header file include path specification.	'-I'
	'OutputFlag'	Required	Prefix name of tool output (object file).	'-o'
	'PreprocessFile'	Required	Output preprocessed source instead of completing compilation.	'-E'
	'PreprocessorDefine'	Required	Prefix preprocessor macro definitions.	'-D'
	'CommandFile'	Optional	Specify command, subcommand, or response file to circumvent command interpreter line length limitations.	'@'
	'GenerateDependencyFiles'	Optional	Generate file describing header file dependencies.	'-MMD'
	'Preinclude'	Optional	Prefix name of header file to include (as if source code is included using #include).	'-include'
	'PreprocessorUndefine'	Optional	Prefix undefinition of a previously defined preprocessor macro.	'-U'
'Archiver'	'Debug'	Required	Generate debugging information.	' '
	'LibrarySearchPath'	Required	Prefix library search path specification.	'-L'
	'OutputFlag'	Required	Prefix name of tool output (static library).	'-o'
	'CommandFile'	Optional	Specify command, subcommand, or response file to circumvent command interpreter line length limitations.	'@'

Build Tool Type	Directive Name (Command-Line Flag)	Optional or Required	Directive Purpose	Example Value
	'Input'	Optional	Prefix name of input object files.	'-input=file.obj'
'Linker' or 'C++ Linker'	'Debug'	Required	Generate debugging information.	'-g'
	'Library'	Required	Prefix "system library".	'-l'
	'LibrarySearchPath'	Required	Prefix library search path specification.	'-L'
	'OutputFlag'	Required	Prefix name of tool output (executable file or shared library).	'-o'
	'CommandFile'	Optional	Specify command, subcommand, or response file to circumvent command interpreter line length limitations.	'@'
	'DefFile'	Optional	Prefix name of module definition file containing information about symbols export (for shared library generation).	'-def:'
	'Input'	Optional	Prefix name of input object files when using a command, subcommand, or response file.	'-input=file.obj'
	'LibraryGroup'	Optional	Encapsulate group of input libraries or object files to indicate multiple linker passes to resolve circular or out-of-order dependencies.	'-Wl,--start-group -Wl,--end-group'
	'LibraryLinkObj'	Optional	Prefix name of input library files.	'-library=file.lib'
	'LibrarySearchPathRuntime'	Optional	Prefix run-time library search path specification.	'-rpath'
	'Shared'	Optional	Indicate that shared library (instead of standalone executable file) must be generated.	'-shared -Wl,--no-undefined'
'Make Tool'	'File'	Required	Prefix name of make file to process.	'-f'

Example: `cCompiler.setDirective('CommandFile', '@');`

directiveValue — Directive value

character vector | string

New value for named directive.

Example: `cCompiler.setDirective('CommandFile', '@');`**pairedDirectiveName — Paired directive name**

character vector | string

Name of directive that contains a pair of command-line flags.

Example: `cLinker.setDirective('LibraryGroup', '-Wl,--start-group', '-Wl,--end-group');`**startValue — Start value**

character vector | string

New start value for paired directive.

Example: `cLinker.setDirective('LibraryGroup', '-Wl,--start-group', '-Wl,--end-group');`**endValue — End value**

character vector | string

New end value for paired directive.

Example: `cLinker.setDirective('LibraryGroup', '-Wl,--start-group', '-Wl,--end-group');`

Examples

Use setDirective to Specify Compiler Command Files

This code snippet shows how you can use the `setDirective` method to state that the C compiler uses the `@` symbol as the prefix to specify a command, subcommand, or response file (to circumvent command interpreter line length limitations).

```
cCompiler = target.create('BuildTool', 'C Compiler', 'gcc', ...  
    'Name', 'MinGW GCC C Compiler');  
cCompiler.setDirective('CommandFile', '@');  
cCompiler.setFileExtensions('Object', {'.obj'});
```

Version History

Introduced in R2023a

See Also

`getDirective` | `showDirectives`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

setFileExtensions

Class: target.BuildTool

Namespace: target

Set file extensions

Syntax

```
myBuildTool.setFileExtensions(fileName,extensionValues)
```

Description

`myBuildTool.setFileExtensions(fileName,extensionValues)` sets the file extensions for the named file type to the specified value.

If the `FileExtensions` property does not contain the corresponding `target.FileExtensionSet` object, the method appends a new object.

If the file type name is not listed in the `target.BuildToolType` object, the method produces an error.

Input Arguments

fileName — File type

character vector | string

Name of file type.

Example: `cCompiler.setFileExtensions('Source', '.cx');`

extensionValues — Extension values

cell array of character vectors | string array

New value for file extensions.

Example: `cCompiler.setFileExtensions('Header', {' .hx'});`

Examples

Use setFileExtensions to Specify Object File Extensions

This code snippet shows you can use the `setFileExtensions` method to specify a C compiler that generates object files with the `.obj` extension.

```
cCompiler = target.create('BuildTool', 'C Compiler', 'gcc', ...
    'Name', 'MinGW GCC C Compiler');
```

```
cCompiler.setDirective('CommandFile', '@');  
cCompiler.setFileExtensions('Object', {'.obj'});
```

Version History

Introduced in R2023a

See Also

[getFileExtensionSet](#) | [showFileExtensions](#)

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

showDirectives

Class: target.BuildTool

Namespace: target

Print directives to screen

Syntax

```
myBuildTool.showDirectives()
```

Description

`myBuildTool.showDirectives()` prints to screen a summary of the configured directives contained in the `target.BuildTool` object.

Version History

Introduced in R2023a

See Also

`getDirective` | `setDirective`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

showFileExtensions

Class: target.BuildTool

Namespace: target

Print file extensions to screen

Syntax

```
myBuildTool.showFileExtensions()
```

Description

`myBuildTool.showFileExtensions()` prints to screen a summary of the configured file extensions contained in the `target.BuildTool` object.

Version History

Introduced in R2023a

See Also

`getFileExtensionSet` | `setFileExtensions`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.BuildToolType Class

Namespace: target

Describe build tool type

Description

The software provides `target.BuildToolType` objects that describe build tool types.

You cannot create `target.BuildToolType` objects.

Properties

Id — Object identifier

string

Unique ID of object in internal database.

Attributes:

GetAccess	public
SetAccess	private

Name — Build tool type

string

Name of build tool type. The software provides objects with these names:

- Assembler
- C Compiler
- C++ Compiler
- Linker
- C++ Linker
- Archiver
- Make Tool
- Delete
- Display
- Move
- HEX Converter
- BIN Converter
- Debug Symbol Converter

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2023a

See Also

`target.BuildTool` | `target.Directive` | `target.FileType` | `target.MakefileBuilder` |
`target.MakeToolType` | `target.PairedDirective` | `target.RepeatingDirective` |
`target.Toolchain`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.CMake Class

Namespace: target

Specify CMake installation for building generated code

Description

Use a `target.CMake` object when you want to define a toolchain that uses a specific CMake installation instead of the version that ships with the software.

To create a `target.CMake` object, use the `target.create` function.

Properties

Version — CMake version

`string`

Optional. Version of the CMake tool.

Attributes:

GetAccess	public
SetAccess	public

Command — Path to CMake executable file

`target.Command` object

An object that describes the full path to the CMake executable file.

Attributes:

GetAccess	public
SetAccess	public

HostOperatingSystemSupport — Platforms supported

`target.HostOperatingSystemSupport` object

An object that gives the platforms that the CMake installation can run on.

Attributes:

GetAccess	public
SetAccess	public

SupportedHardware — Target devices

`target.HardwareComponentSupport` object array

Optional. An object that gives the target devices that the CMake installation can build code for. If the property is empty, it is treated as a wildcard.

Attributes:

GetAccess	public
SetAccess	public

Name — CMake tool name

string

A unique name for your CMake tool installation.

Example: 'CMake with Third-Party Customizations'

Attributes:

GetAccess	public
SetAccess	public

Examples

Use target.CMake to Specify CMake Installation

This code snippet shows how you can use the `Tools` property of the `target.Toolchain` object and a `target.CMake` object to specify your CMake installation.

```
tc = target.create('Toolchain', 'Name', 'CMake/Ninja for all hosts');
tc.Builder = target.create('CMakeBuilder');
tc.Builder.Generator = 'Ninja';
tc.Builder.CommandLineCacheEntries(end+1) = ...
    target.create('CMakeCacheEntry', 'Name', 'CMAKE_C_COMPILER', 'Value', 'gcc');
tc.Builder.CommandLineCacheEntries(end+1) = ...
    target.create('CMakeCacheEntry', 'Name', 'CMAKE_CXX_COMPILER', 'Value', 'g++');

tc.Tools = target.create('CMake', 'Name', 'My CMake')
tc.Tools(1).Command.String = 'C:\pathToMyCMakeInstallation\cmake.exe'
```

Version History

Introduced in R2022b

See Also

`target.Toolchain` | `target.CMakeBuildType` | `target.CMakeBuilder` |
`target.CMakeCacheEntry` | `target.EnvironmentConfiguration` |
`target.HardwareComponentSupport` | `target.create`

Topics

"Create Custom CMake Toolchain Definition"

target.CMakeCacheEntry Class

Namespace: target

Configure a CMake cache entry

Description

Use a `target.CMakeCacheEntry` object to define a CMake cache entry for the CMake tool. The software adds the cache entry to the command that invokes CMake, using one of these forms:

- `-D Name:Type=Value`
- `-D Name=Value`

Name, Type, and Value are object properties.

To create a `target.CMakeCacheEntry` object, use the `target.create` function.

Properties

Name — Cache entry name

string

Name of the CMake cache entry.

If the string value has spaces, for example, `name With Spaces`, the software passes to the CMake tool the value with double quotes, that is, `"name With Spaces"`.

Attributes:

GetAccess	public
SetAccess	public

Type — Cache entry type

string

Optional. Type of the CMake cache entry.

If the string value has spaces, for example, `name With Spaces`, the software passes to the CMake tool the value with double quotes, that is, `"name With Spaces"`.

Attributes:

GetAccess	public
SetAccess	public

Value — Cache entry value

string

Value of the CMake cache entry.

If the string value has spaces, for example, `name With Spaces`, the software passes to the CMake tool the value with double quotes, that is, `"name With Spaces"`.

Attributes:

GetAccess	public
SetAccess	public

Examples**Define CMake Cache Entries for CMake Tool**

This code snippet shows how you can use the `target.CMakeCacheEntry` object to define CMake cache entries for the CMake tool.

```
tc = target.create('Toolchain', 'Name', 'CMake/Ninja for all hosts');
tc.Builder = target.create('CMakeBuilder');
tc.Builder.Generator = 'Ninja';
tc.Builder.CommandLineCacheEntries(end+1) = ...
    target.create('CMakeCacheEntry', 'Name', 'CMAKE_C_COMPILER', 'Value', 'gcc');
tc.Builder.CommandLineCacheEntries(end+1) = ...
    target.create('CMakeCacheEntry', 'Name', 'CMAKE_CXX_COMPILER', 'Value', 'g++');
```

Version History

Introduced in R2022b

See Also

`target.Toolchain` | `target.CMake` | `target.CMakeBuildType` | `target.CMakeBuilder` |
`target.EnvironmentConfiguration` | `target.HardwareComponentSupport` |
`target.create`

Topics

“Create Custom CMake Toolchain Definition”

target.CMakeBuilder Class

Namespace: target

Configure how CMake builds generated code

Description

Use a target.CMakeBuilder object to configure how CMake builds generated code.

To create a target.CMakeBuilder object, use the `target.create` function.

Properties

CommandLineCacheEntries — Cache entries

target.CMakeCacheEntry object array

Optional. Array of additional CMake cache entries to pass to the CMake tool as part of the command line invocation. The software expands the entries in this form:

```
cmake -D name:type=value ...
```

Attributes:

GetAccess	public
SetAccess	public

Generator — CMake generator

string

Generator that CMake must use, that is, the *generator* value for the command `cmake -G generator`.

If the string value has no spaces, for example, `nameWithNoSpaces`, the software passes this command to the CMake tool:

```
cmake -G nameWithNoSpaces
```

If the string value has spaces, for example, `name With Spaces`, the software passes this command to the CMake tool:

```
cmake -G "name With Spaces"
```

Attributes:

GetAccess	public
SetAccess	public

GeneratorIsMultiConfig — Multi-configuration generator

true | false

Whether generator is multi-configuration:

- `true` -- CMake generator supports multiple configurations.
- `false` -- CMake generator supports single configuration

Attributes:

GetAccess	public
SetAccess	public

Data Types: `logical`

PlatformName — Platform name

`string`

Optional. Platform name for the generator, that is, the *platformName* value for the command `cmake -A platformName`.

If the string value has no spaces, for example, `nameWithNoSpaces`, the software passes this command to the CMake tool:

```
cmake -A nameWithNoSpaces
```

If the string value has spaces, for example, `name With Spaces`, the software passes this command to the CMake tool:

```
cmake -A "name With Spaces"
```

Attributes:

GetAccess	public
SetAccess	public

SupportedBuildTypes — Supported build types

`target.CMakeBuildType` object array

Build types or build configurations supported by generator and toolchain file.

The property is prepopulated with definitions of build types and build configurations that are available by default. If the `Generator` property is set to `Ninja Multi-Config`, the build type `MinSizeRel`, is not supported by default. Unless you define `MinSizeRel` in a toolchain file, you must remove it from the object array. This code snippet shows how you can remove `MinSizeRel`.

```
tc = target.create('Toolchain', 'Name', 'CMake/Ninja Multi-Config');
tc.Builder = target.create('CMakeBuilder');
tc.Builder.Generator = 'Ninja Multi-Config';
tc.Builder.GeneratorIsMultiConfig = true;
tc.Builder.CommandLineCacheEntries(end+1) = ...
    target.create('CMakeCacheEntry', 'Name', 'CMAKE_C_COMPILER', 'Value', 'gcc');
tc.Builder.CommandLineCacheEntries(end+1) = ...
    target.create('CMakeCacheEntry', 'Name', 'CMAKE_CXX_COMPILER', 'Value', 'g++');
tc.Builder.SupportedBuildTypes(strcmp({tc.Builder.SupportedBuildTypes}, 'MinSizeRel')) = [];
```

Attributes:

GetAccess	public
SetAccess	public

ToolchainFile — Path to toolchain file

`string`

Optional. Path to a CMake toolchain file, that is, the *toolChain* value for the command `cmake -D CMAKE_TOOLCHAIN_FILE=toolChain`.

If the string value has no spaces, for example, `nameWithNoSpaces`, the software passes this command to the CMake tool:

```
cmake -D CMAKE_TOOLCHAIN_FILE=nameWithNoSpaces
```

If the string value has spaces, for example, `name With Spaces`, the software passes this command to the CMake tool:

```
cmake -D CMAKE_TOOLCHAIN_FILE="name With Spaces"
```

Attributes:

GetAccess	public
SetAccess	public

ToolsetSpecification — Tool set specification

string

Optional. Tool set specification for generator, that is, the *toolSet* value for the command `cmake -T toolSet`.

If the string value has no spaces, for example, `nameWithNoSpaces`, the software passes this command to the CMake tool:

```
cmake -T nameWithNoSpaces
```

If the string value has spaces, for example, `name With Spaces`, the software passes this command to the CMake tool:

```
cmake -T "name With Spaces"
```

Attributes:

GetAccess	public
SetAccess	public

Examples

Use target.CMakeBuilder to Configure CMake Build

This code snippet from “Create Custom CMake Toolchain Definition” shows how you can use the `Builder` property of the `target.Toolchain` object and a `target.CMakeBuilder` object to configure how CMake builds generated code.

```
tc = target.create('Toolchain', 'Name', 'Example Custom CMake Toolchain');
tc.Builder = target.create('CMakeBuilder');
tc.Builder.Generator = 'Ninja';
tc.Builder.ToolchainFile = fullfile(pwd, 'ExampleCMakeToolchain.cmake');
tc.Builder.SupportedBuildTypes(end+1) = ...
    target.create('CMakeBuildType', ...
        'Name', 'FastMath', ...
        'GeneratesDebugSymbols', false, ...
        'DebugBuildType', 'FastMathWithDebug');
tc.Builder.SupportedBuildTypes(end+1) = ...
    target.create('CMakeBuildType', ...
```

```
'Name', 'FastMathWithDebug', ...  
'GeneratesDebugSymbols', true);
```

Version History

Introduced in R2022b

See Also

[target.Toolchain](#) | [target.CMake](#) | [target.CMakeBuildType](#) | [target.CMakeCacheEntry](#) |
[target.EnvironmentConfiguration](#) | [target.HardwareComponentSupport](#) |
[target.create](#)

Topics

[“Create Custom CMake Toolchain Definition”](#)

target.CMakeBuildType Class

Namespace: target

Describe CMake build type or build configuration

Description

Use a `target.CMakeBuildType` object to describe a CMake build type (for single configuration generators) or build configuration (for multi-configuration generators) that is supported by a given combination of generator and CMake toolchain file. When you select the associated toolchain for the **Toolchain** configuration parameter, the build type is available as a **Build Configuration** option.

To create a `target.CMakeBuildType` object, use the `target.create` function.

Properties

Name — Build type name

string

Name of CMake build type. For example, Debug, Release, RelWithDebInfo, and MinSizeRel.

Attributes:

GetAccess	public
SetAccess	public

Data Types: string

GeneratesDebugSymbols — Debug symbol generation

true | false

Whether the build type supports generation of debug symbols:

- `true` -- Supports generation of debug symbols
- `false` -- Does not support generation of debug symbols

Attributes:

GetAccess	public
SetAccess	public

Data Types: logical

DebugBuildType — Substitute build type

string

Substitute CMake build type for debug workflow. When a workflow requires debug symbols but the current CMake build type does not support the generation of debug symbols, use the property to provide a substitute build type that supports a debug workflow.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify CMake Build Type

This code snippet from “Create Custom CMake Toolchain Definition” shows how you can use the `SupportedBuildTypes` property of a `target.CMakeBuilder` object and a `target.CMakeBuildType` object to specify CMake build types for a combination of generator and CMake toolchain file.

```
tc = target.create('Toolchain', 'Name', 'Example Custom CMake Toolchain');
tc.Builder = target.create('CMakeBuilder');
tc.Builder.Generator = 'Ninja';
tc.Builder.ToolchainFile = fullfile(pwd, 'ExampleCMakeToolchain.cmake');
tc.Builder.SupportedBuildTypes(end+1) = ...
    target.create('CMakeBuildType', ...
        'Name', 'FastMath', ...
        'GeneratesDebugSymbols', false, ...
        'DebugBuildType', 'FastMathWithDebug');
tc.Builder.SupportedBuildTypes(end+1) = ...
    target.create('CMakeBuildType', ...
        'Name', 'FastMathWithDebug', ...
        'GeneratesDebugSymbols', true);
```

Version History

Introduced in R2022b

See Also

`target.Toolchain` | `target.CMake` | `target.CMakeBuilder` | `target.CMakeCacheEntry` |
`target.EnvironmentConfiguration` | `target.HardwareComponentSupport` |
`target.create`

Topics

“Create Custom CMake Toolchain Definition”

target.Command Class

Namespace: target

Capture system command for execution on MATLAB computer

Description

Use the `target.Command` class to capture a system command for execution on your development computer.

To create a `target.Command` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, create the object and specify properties in a single step.

```
commandObject = target.create('Command', ...
                             stringPropertyValue, ...
                             argumentsPropertyValue)
```

Properties

String — Command string

string

Name of the application or script to call.

Attributes:

GetAccess	public
SetAccess	public

Arguments — Application or script arguments

string array | cell array of character vectors

String array or cell array of character vectors where each element represents a separate argument to the application or script defined in the `String` property.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create target.Command Objects

Create this `target.Command` object by providing an application file path and arguments for the application.

```
cmdObj = target.create('Command');
cmdObj.String = 'pathToavrdude';
cmdObj.Arguments = {'-p$(BOARD.Processor.ArduinoPartNumber)' ...
                  '-c$(PROCESSOR.ArduinoProgrammer)' ...
                  '-Uflash:w:$(EXE):i'};
```

You can create the object in a single step.

```
cmdObj = target.create('Command', ...  
    'pathToavrdude', ...  
    {'-p$(BOARD.Processor.ArduinoPartNumber)' ...  
    '-c$(PROCESSOR.ArduinoProgrammer)' ...  
    '-Uflash:w:$(EXE):i'});
```

You can also specify the command and arguments by using a string. For example, to create a `target.Command` object for the command `echo` with arguments `-a` and `-b`, run:

```
cmdObj = target.create('Command', 'echo -a -b');
```

Version History

Introduced in R2020b

See Also

`target.create` | `target.ApplicationExecutionTool` |
`target.HostProcessExecutionTool` | `target.SystemCommandExecutionTool`

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.CommunicationChannel Class

Namespace: target

Describe communication channel properties

Description

Use a `target.CommunicationChannel` object to describe communication channel properties for an I/O connection between two systems. You can use the object as part of a `target.Connection` object. `target.RS232Channel`, `target.TCPChannel`, and `target.UDPChannel` are examples of predefined communication channels.

Properties

Name — `target.CommunicationChannel` object name

character vector | string

Name of `target.CommunicationChannel` object.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

Examples

Create Connection by Using TCP Communication Channel

This code from “Set Up PIL Connectivity by Using Target Framework” shows how to specify the connection between your development computer and target hardware. In the example, the target application runs on your development computer as a separate process and uses a TCP communication channel through `localhost`.

```
connection = target.create('TargetConnection');
connection.Name = 'Host Process Connection';
connection.Target = hostTarget;
connection.CommunicationChannel = target.create('TCPChannel');
connection.CommunicationChannel.Name = ...
    'External Process TCPCommunicationChannel';
connection.CommunicationChannel.IPAddress = 'localhost';
connection.CommunicationChannel.Port = '0';
```

Note Using name-value arguments, you can create the connection object with this command:

```
connection = target.create('TargetConnection', ...
    'Name', 'Host Process Connection', ...
    'Target', hostTarget, ...
    'CommunicationType', 'TCPChannel', ...
```

```
'IPAddress', 'localhost', ...  
'Port', '0')
```

Version History

Introduced in R2020b

See Also

`target.create` | `target.RS232Channel` | `target.TCPChannel` | `target.UDPChannel`

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.CommunicationInterface Class

Namespace: target

Describe data I/O details for target hardware

Description

Use the `target.CommunicationInterface` class to describe data transfer for your target hardware. Associate the communication channel for data transfer and the device driver API implementation with a `target.CommunicationInterface` object.

Properties

Name — `target.CommunicationInterface` object name

character vector | string

Name of `target.CommunicationInterface` object.

Attributes:

GetAccess	public
SetAccess	public

Channel — Communication channel

character vector | string

Type of communication channel that connects to the target hardware. For example, if you define PIL connectivity by using a `target.TargetConnection` over a `target.RS232Channel`, set this property to 'RS232Channel'.

Attributes:

GetAccess	public
SetAccess	public

APIImplementations — Device driver API implementations for communication interface

`target.APIImplementations` object array

Details of the API implementations that use the communication interface channel to support data transfer to and from the target hardware. For example, an `rtiostream` API implementation for PIL connectivity.

Attributes:

GetAccess	public
SetAccess	public

Examples

Communication Interface for Target Hardware

Create the communication interface for the target hardware. This code snippet from “Set Up PIL Connectivity by Using Target Framework” shows how to create the interface.

```
comms = target.create('CommunicationInterface');
comms.Name = 'Linux TCP Interface';
comms.Channel = 'TCPChannel';
comms.APIImplementations = target.create('APIImplementation', ...
    'Name', 'x86_rtiostream Implementation');
comms.APIImplementations.API = target.create('API', 'Name', 'rtiostream');
comms.APIImplementations.BuildDependencies = target.create('BuildDependencies');
comms.APIImplementations.BuildDependencies.SourceFiles = ...
    {fullfile('${MATLABROOT}', ...
        'toolbox', ...
        'coder', ...
        'rtiostream', ...
        'src', ...
        'rtiostreamtcpip', ...
        'rtiostream_tcpip.c')};
comms.APIImplementations.MainFunction = target.create('MainFunction', ...
    'Name', 'TCP RtIOStream Main');
comms.APIImplementations.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
hostTarget.CommunicationInterfaces = comms;
```

Version History

Introduced in R2020b

See Also

[target.create](#) | [target.CommunicationChannel](#) | [target.APIImplementation](#)

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.CommunicationProtocolStack Class

Namespace: target

Describe communication protocol parameters

Description

To describe target-specific parameters for different protocols, you can associate a `target.CommunicationProtocolStack` object with a `target.Board` object. One example of `target.CommunicationProtocolStack` is `target.PILProtocol`.

To create a `target.CommunicationProtocolStack` object, use the `target.create` function.

Properties

Name — API name

character vector | string

Name of the API.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify PIL Communication Protocol Object for `target.Board`

Specify PIL protocol information. This code snippet from “Set Up PIL Connectivity by Using Target Framework” shows how to specify the information.

```
pilProtocol = target.create('PILProtocol');
pilProtocol.Name = 'Linux PIL Protocol';
pilProtocol.SendBufferSize = 50000;
pilProtocol.ReceiveBufferSize = 50000;
hostTarget.CommunicationProtocolStacks = pilProtocol;
```

Version History

Introduced in R2020b

See Also

`target.create` | `target.PILProtocol`

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.Connection Class

Namespace: target

Base class for target connection properties

Description

Capture target connection properties by using a class derived from `target.Connection`. For example, use the `target.TargetConnection` class to describe a target connection from the development computer, inheriting properties from the `target.Connection` class.

Class Attributes

Abstract	true
HandleCompatible	true

For information on class attributes, see “Class Attributes”.

Properties

Name — Connection object

character vector | string

Name of the connection object.

Attributes:

GetAccess	public
SetAccess	private

CommunicationChannel — Communication channel description

`target.CommunicationChannel` object

Use this property to associate the connection with a specific `target.CommunicationChannel` that is used to determine common connection properties between the connected systems. For example, `target.RS232Channel`.

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2020b

See Also

`target.create` | `target.TargetConnection` | `target.CommunicationChannel`

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.connection.close

Namespace: target.connection

Close connection between MATLAB development computer and remote build computer

Syntax

```
target.connection.close(boardName)
```

Description

`target.connection.close(boardName)` closes the connection between your MATLAB development computer and the remote build computer.

Examples

Close Connection

For an example that uses `target.connection.close`, see Build Generated Code on Raspberry Pi Computer.

Input Arguments

boardName — Board name

string

Required. Name of target hardware board, which must correspond to the `Name` property of the `target.Board` object that provides MATLAB with a description of the remote build computer.

Example: `target.connection.close(boardName);`

Version History

Introduced in R2024a

See Also

`target.connection.open` | `target.connection.show`

Topics

Build Generated Code Remotely

target.connection.open

Namespace: target.connection

Create a connection between MATLAB development computer and remote build computer

Syntax

```
target.connection.open(boardName, type)
target.connection.open( ___, Name=Value)
```

Description

`target.connection.open(boardName, type)` creates a connection between your MATLAB development computer and a remote build computer.

`target.connection.open( ___, Name=Value)` creates a connection using one or more name-value arguments.

Examples

Open Connection

For an example that uses `target.connection.open`, see [Build Generated Code on Raspberry Pi Computer](#).

Input Arguments

boardName — Board name

string

Required. Name of target hardware board, which must correspond to the `Name` property of a `target.Board` object in the internal database. The `target.Board` object provides MATLAB with a description of the remote build computer.

Example: `target.connection.open(boardName, 'SSH', IP='192.168.0.10', Username='pi');`

type — Connection type

'SSH' (default)

Optional. Type of connection between MATLAB and remote build computers. Only Secure Shell (SSH) is supported.

Example: `target.connection.open(boardName, 'SSH', IP='192.168.0.10', Username='pi');`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `target.connection.open(boardName, 'SSH', IP='remoteHost.com', Port=22, Username='myName', AuthenticationMode='Password');`

IPAddress — IP address

character vector | string

Required. IP address of remote build computer.

Port — Port number

22 (default) | integer scalar

Optional. Port number for connection type.

Username — User name

character vector | string

Optional. User name.

AuthenticationMode — SSH authentication mode

'Password' (default) | 'IdentityFile'

Optional. Mode of authentication for SSH access to remote build computer.

IdentityFilePath — SSH identity file path

"" (default) | string

Optional. Path to SSH identity file. If you specify this option, the software sets `AuthenticationMode` to 'IdentityFile' internally. Use this option if you want to avoid interactive selection of an identify file via the GUI.

RequiresPassphrase — SSH passphrase flag

false (default) | true

Optional. Specify whether the SSH identity file is password-locked:

- `true` -- Access to identify file requires password.
- `false` -- Access to identity file does not require password.

Version History

Introduced in R2024a

See Also

`target.connection.show` | `target.connection.close`

Topics

Build Generated Code Remotely

target.connection.show

Namespace: target.connection

Show connection between MATLAB development computer and remote build computer

Syntax

```
target.connection.show()
```

Description

`target.connection.show()` shows the active connection between your MATLAB development computer and the remote build computer.

Examples

Show Connection

For an example that uses `target.connection.show`, see Build Generated Code on Raspberry Pi Computer.

Version History

Introduced in R2024a

See Also

`target.connection.open` | `target.connection.close`

Topics

Build Generated Code Remotely

target.ConnectionProperties Class

Namespace: target

Describe target-specific connection properties

Description

Use a `target.ConnectionProperties` object to describe connection properties that are required for the target hardware to connect to another system. The class is used as a base class for specific connection properties such as `target.Port`. You can extend the class to provide custom properties by using the `target.AddOn` class.

Properties

AddOns — Custom property add-on objects

`target.AddOns` object array

To provide additional custom properties, associate one or more `target.AddOn` objects with the `target.ConnectionProperties` object.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Methods

Public Methods

<code>get</code>	Get name of custom add-on property
<code>set</code>	Set property value

Version History

Introduced in R2020b

See Also

`target.create`

get

Class: target.ConnectionProperties

Namespace: target

Get name of custom add-on property

Syntax

```
value = get(propertyName)
```

Description

`value = get(propertyName)` returns the name of a custom add-on property or `target.ConnectionProperties` property.

Input Arguments

propertyName — Property name

string

Name of a custom add-on property or `target.ConnectionProperties` class property.

Output Arguments

Value — Property value

string

Property value that corresponds to the `propertyName`.

Version History

Introduced in R2020b

See Also

`target.ConnectionProperties`

set

Class: target.ConnectionProperties

Namespace: target

Set property value

Syntax

```
set(propertyName,propertyValue)
```

Description

set(propertyName,propertyValue) sets the name of a custom add-on property or target.ConnectionProperties property to a specified value.

Input Arguments

propertyName — Property name

string

Name of a custom add-on property or target.ConnectionProperties class property.

propertyValue — Property value

string

Value for propertyName.

Version History

Introduced in R2020b

See Also

target.ConnectionProperties

target.clear

Namespace: target

Clear all target objects from internal database

Syntax

```
target.clear()
target.clear(Name,Value)
```

Description

`target.clear()` clears target objects from an internal database. The function clears only objects added to the database in the current MATLAB session.

`target.clear(Name,Value)` clears target objects from the internal database using `Name,Value` arguments. For example, you can clear persistent objects added in previous MATLAB sessions.

Examples

Clear Target Object Data From Internal Database

This example shows how you can use `target.clear` to clear target object data from the internal database.

Create an array of target objects.

```
langImp = target.create('LanguageImplementation', ...
    'Name', 'LanguageImplementation1', ...
    'Copy', 'Atmel-AVR');

proc1 = target.create('Processor', 'Name', 'Processor1');
proc2 = target.create('Processor', 'Name', 'Processor2');

board1 = target.create('Board', 'Name', 'Board1');

proc2.LanguageImplementations = langImp;
board1.Processors = proc2;
objects = [board1 proc1 proc2 langImp];
```

For the current MATLAB session, add the objects to the internal database.

```
target.add(objects);
```

To clear the added objects from the internal database, run:

```
target.clear()
```

The command `target.clear()` is equivalent to `target.clear(MATLABSession=true)`.

For persistence over MATLAB sessions, add the objects to the internal database by using the `UserInstall=true` name-value argument:

```
target.add(objects, UserInstall=true);
```

To clear the persistent objects from the internal database, run:

```
target.clear(UserInstall=true)
```

When you clear "User Install", persistent data will be removed permanently from an internal database. Enter Y (default) to continue or N to stop.

If you enter Y, the function removes the persistent objects from the internal database.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `target.clear(UserInstall=true);`

MATLABSession — Current session objects

true (default) | false

Specify removal of target objects added to the internal database in current MATLAB session:

- `true` -- Remove objects added in current session.
- `false` -- Do not remove objects added in current session.

Example: `target.clear(MATLABSession=true);`

Data Types: logical

UserInstall — Persistent objects

false (default) | true

Specify removal of persistent target objects added to the internal database in current and previous MATLAB sessions:

- `true` -- Remove objects added in current and previous sessions.
- `false` -- Do not remove objects added in previous sessions.

Example: `target.clear(UserInstall=true);`

Data Types: logical

Version History

Introduced in R2023a

See Also

`target.add` | `target.create` | `target.remove` | `target.update`

Topics

"Register New Hardware Devices"

target.create

Namespace: target

Create target object

Syntax

```
targetObject = target.create(targetType)
targetObject = target.create(targetType,Name,Value)
```

Description

`targetObject = target.create(targetType)` creates and returns an object of the specified class.

`targetObject = target.create(targetType,Name,Value)` configures the object using one or more name-value arguments.

Note You can create an object and specify properties in one step for these classes:

- `target.BuildTool`
 - `target.Command`
 - `target.MakefileBuilder`
 - `target.TargetConnection`
 - `target.Timer`
 - `target.Toolchain`
-

Examples

Specify Hardware Implementation for New Device

This example shows how to register a new hardware device.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create("Processor",Name="MyProcessor", ...
                        Manufacturer="MyManufacturer");
```

Create a `target.LanguageImplementation` object for language implementation details.

```
myLanguageImplementation = target.create("LanguageImplementation", ...
                                           Name="MyProcessorImplementation");
```

Specify language implementation details.

```
myLanguageImplementation.Endianness = target.Endianness.Little;
```

```
myLanguageImplementation.AtomicIntegerSize = 64;
myLanguageImplementation.AtomicFloatSize = 64;
myLanguageImplementation.WordSize = 64;

myLanguageImplementation.DataTypes.Char.Size = 8;
myLanguageImplementation.DataTypes.Short.Size = 16;
myLanguageImplementation.DataTypes.Int.Size = 32;
myLanguageImplementation.DataTypes.Long.Size = 64;
myLanguageImplementation.DataTypes.LongLong.IsSupported = true;
myLanguageImplementation.DataTypes.LongLong.Size = 64;
myLanguageImplementation.DataTypes.Float.Size = 32;
myLanguageImplementation.DataTypes.Double.Size = 64;

myLanguageImplementation.DataTypes.Pointer.Size = 32;

myLanguageImplementation.DataTypes.SizeT.Size = 64;
myLanguageImplementation.DataTypes.PtrDiffT.Size = 64;
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myLanguageImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

```
target.add summary:
```

```
Objects added to internal database for current MATLAB session:
target.LanguageImplementation    "MyProcessorImplementation"
target.Processor              "MyManufacturer-MyProcessor"
```

- If you are using the MATLAB® Coder™: On the **Hardware** tab, you see the new device. Alternatively, you can now create a `coder.Hardware` object for this device by using the `coder.hardware` function.
- If you are using the Simulink® Coder™: On the **Hardware Implementation** pane, you can now set **Device vendor** and **Device type** to `MyManufacturer` and `MyProcessor` respectively.

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded)
```

```
target.remove summary:
```

```
Objects removed from internal database:
target.LanguageImplementation    "MyProcessorImplementation"
target.Processor              "MyManufacturer-MyProcessor"
```

Create Hardware Implementation by Modifying Existing Implementation

If an existing hardware implementation contains most of the values that you want in a new hardware implementation, you can quickly create the new implementation by creating and modifying a copy of the existing implementation.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create("Processor",Name="MyProcessor", ...
    Manufacturer="MyManufacturer");
```

Create a `target.LanguageImplementation` object that copies an existing language implementation.

```
myCopiedImplementation = target.create("LanguageImplementation", ...
    Name="MyCopiedImplementation", ...
    Copy="Atmel-AVR");
```

Specify the required language implementation details. For example, byte ordering.

```
myCopiedImplementation.Endianness = target.Endianness.Big;
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myCopiedImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

`target.add` summary:

```
Objects added to internal database for current MATLAB session:
target.LanguageImplementation    "MyCopiedImplementation"
target.Processor               "MyManufacturer-MyProcessor"
```

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded)
```

`target.remove` summary:

```
Objects removed from internal database:
target.LanguageImplementation    "MyCopiedImplementation"
target.Processor               "MyManufacturer-MyProcessor"
```

Create Hardware Implementation by Reusing Existing Implementation

If your hardware device requires the same hardware implementation as an existing implementation, you can reuse the existing implementation.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create("Processor",Name="MyProcessor", ...
    Manufacturer="MyManufacturer");
```

Retrieve the existing implementation by using the identifier for the device vendor and type.

```
existingImplementation = target.get("LanguageImplementation", ...
    "ARM Compatible-ARM Cortex");
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = existingImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);  
  
target.add summary:  
  
    Objects added to internal database for current MATLAB session:  
    target.Processor          "MyManufacturer-MyProcessor"  
    1 object not added because they already exist.
```

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded);  
  
target.remove summary:  
  
    Objects removed from internal database:  
    target.Processor          "MyManufacturer-MyProcessor"
```

Input Arguments

targetType — Target type

character vector | string

Specify class of object. For example, specifying:

- 'Processor' creates a `target.Processor` object.
- 'LanguageImplementation' creates a `target.LanguageImplementation` object.
- 'Alias' creates a `target.Alias` object.

For the full list of supported types, see `target`.

Example: 'Processor'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `myProc = target.create('Processor', 'Name', 'myProcessor', 'Manufacturer', 'myProcessorManufacturer');`

Copy — Copy existing target feature object

character vector | string

Create a target object by copying values from an existing target object. For example:

```
myLangImp = target.create('LanguageImplementation', ...  
                           'Name', 'myLanguageImplementation', ...  
                           'Copy', 'ARM Compatible-ARM Cortex');
```

IDPrefix — ID prefix

character vector | string

Add a prefix to the IDs generated by `target.create`.

***propertyName* — Property name**

character vector | string

Create the target object with properties that are set to values that you specify.

Output Arguments

targetObject — Target object

object

The object that is created and returned. For example, the object is a:

- `target.Processor` object if `targetType` is `'Processor'`
- `target.LanguageImplementation` object if `targetType` is `'LanguageImplementation'`
- `target.Alias` object if `targetType` is `'Alias'`

Version History

Introduced in R2019a

See Also

`target.add` | `target.get` | `target.remove`

Topics

“Register New Hardware Devices”

target.DebugExecutionTool Class

Namespace: target

Provide MATLAB service interface for debugger to manage processes on target hardware

Description

Use the `target.DebugExecutionTool` class, which inherits from `target.ExecutionTool`, to provide a MATLAB service interface for a debugger. The interface enables the debugger to start and track a process on the target hardware.

Properties

Breakpoints — Command-line arguments

`target.Breakpoint` object array

Command-line arguments to pass to the target application.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

Methods

Public Methods

<code>close</code>	Close target application harness
<code>getApplicationStatus</code>	Get status of target application
<code>getStandardError</code>	Return standard error stream from target application
<code>getStandardOutput</code>	Return standard output stream from target application
<code>loadApplication</code>	Load target application into harness
<code>open</code>	Open target application harness
<code>startApplication</code>	Start execution of target application
<code>stopApplication</code>	Stop execution of target application
<code>unloadApplication</code>	Unload target application from harness

Examples

Provide Debugging for PIL Simulation

For an example that uses the `target.DebugExecutionTool` class, see “Support PIL Debugging” and “DebugExecutionTool Template”.

Version History

Introduced in R2023a

See Also

[target.ExecutionService](#) | [target.ExecutionTool](#)

Topics

[“Set Up PIL Connectivity by Using Target Framework”](#)

target.DebugIOTool Class

Namespace: target

Debug byte stream I/O tool service interface

Description

To provide a service interface for a tool that starts and tracks an application on the target hardware through a debugger, define a subclass that derives from the `target.DebugIOTool` class. The tool controls the debugger's interaction with the executing application and reads and writes to memory through the debugger. The `target.DebugIOTool` class inherits from `target.DebugExecutionTool`.

Properties

Breakpoints — Application breakpoints

`target.Breakpoint` object array

Application breakpoints that you must set in the debugger.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Methods

Public Methods

<code>close</code>	Close target application harness
<code>getApplicationStatus</code>	Get status of target application
<code>getStandardError</code>	Return standard error stream from target application
<code>getStandardOutput</code>	Return standard output stream from target application
<code>loadApplication</code>	Load target application into harness
<code>open</code>	Open target application harness
<code>read</code>	Read specified byte stream from variable in memory
<code>startApplication</code>	Start execution of target application
<code>stopApplication</code>	Stop execution of target application
<code>unloadApplication</code>	Unload target application from harness
<code>write</code>	Write byte stream to variable in memory

Examples

PIL Target Connectivity with Debugger

For an example that uses the `target.DebugIOTool` class, see “Use Debugger for PIL Target Connectivity”.

Version History

Introduced in R2021a

See Also

`target.DebugExecutionTool` | `target.ExecutionService` | `target.ExecutionTool`

Topics

"Use Debugger for PIL Target Connectivity"

"DebugIOTool Template"

read

Class: target.DebugIOTool

Namespace: target

Read specified byte stream from variable in memory

Syntax

```
[variableData, errFlag] = myDebugIOTool.read(byteStream, variable)
```

Description

`[variableData, errFlag] = myDebugIOTool.read(byteStream, variable)`, through the debugger, reads the specified byte stream from the variable in memory. The method returns the byte stream and an error flag.

Input Arguments

byteStream — Byte stream

string

Byte stream to write to memory.

Example: `[vd,ef] = myDebugIOTool.read('myByteStream', myVariable)`

variable — Variable in memory

vector

Variable in memory.

Example: `[vd,ef] = myDebugIOTool.read('myByteStream', myVariable)`

Data Types: uint64

Output Arguments

variableData — Byte stream

vector

Byte stream from variable in memory.

errFlag — Error flag

true | false

Outcome of the write operation:

- `true` -- Error occurred during write operation.
- `false` -- Write operation completed.

Version History

Introduced in R2021a

See Also

`target.DebugIOTool` | `write`

write

Class: target.DebugIOTool

Namespace: target

Write byte stream to variable in memory

Syntax

```
errFlag = myDebugIOTool.write(byteStream, variable)
```

Description

`errFlag = myDebugIOTool.write(byteStream, variable)`, through the debugger, writes the specified byte stream to the variable in memory. The method returns an error flag.

Input Arguments

byteStream — Byte stream

string

Byte stream to write to memory.

Example: `ef = myDebugIOTool.write('myByteStream', myVariable)`

variable — Variable in memory

vector

Destination variable in memory.

Example: `ef = myDebugIOTool.write('myByteStream', myVariable)`

Data Types: `uint64`

Output Arguments

errFlag — Error flag

`true` | `false`

Outcome of the write operation:

- `true` -- Error occurred during write operation.
- `false` -- Write operation completed.

Version History

Introduced in R2021a

See Also

`target.DebugIOTool` | `write`

target.Directive Class

Namespace: target

Describe command-line flag for tool

Description

Use a `target.Directive` object to describe a command-line flag for a build tool.

To create a `target.Directive` object, use the `setDirective` method of the `target.BuildTool` object.

Properties

Name — Directive name

string

Name of directive.

Attributes:

GetAccess	public
SetAccess	public

Value — Directive value

string

Value of directive.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify C Compiler that Uses Command Files

This code snippet shows how you can specify a C compiler that uses command files.

```
cCompiler = target.create('BuildTool', 'C Compiler', 'gcc', ...
    'Name', 'MinGW GCC C Compiler');
cCompiler.setDirective('CommandFile', '@');
```

Version History

Introduced in R2023a

See Also

`target.BuildTool` | `target.BuildToolType` | `target.FileType` | `target.MakefileBuilder` | `target.MakeToolType` | `target.PairedDirective` | `target.RepeatingDirective` | `target.Toolchain`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.Emulator Class

Namespace: target

Provide target emulator details

Description

Use a `target.Emulator` object to provide MATLAB with data about your target emulator, for example, emulator launch method, and emulated hardware information.

To create a `target.Emulator` object, use the `target.create` function.

Properties

Name — Emulator name

character vector | string

Name of the `target.Emulator` object, which `target.get` uses as an identifier in data retrieval.

Attributes:

GetAccess	public
SetAccess	public

Target — Emulated hardware board

`target.Board` object

`target.Board` object that provides hardware board details.

Attributes:

GetAccess	public
SetAccess	public

LaunchTool — Tool to start emulator

`target.ApplicationExecutionTool` object

Object that provides descriptions of supported tools for starting emulator application on your development computer.

Attributes:

GetAccess	public
SetAccess	public

Examples

Use target.Emulator Object to Set Up Target Connectivity

This example shows how to use a `target.Emulator` object to set up target connectivity.

Create a board object to use with the emulator.

```
emulatorBoard = target.create("Board", ...  
    Name="ARM Cortex-A Emulator-Based Test Board");  
Processors=target.get("Processor", "ARM Compatible-ARM Cortex");
```

Create a `target.Emulator` object and associate it with the emulated target board.

```
emulatorTarget = target.create("Emulator", ...  
    Name="Example Emulator Target");  
emulatorTarget.Target = emulatorBoard;
```

Create a `target.HostProcessExecutionTool` object.

```
hostProcLaunchTool = target.create("HostProcessExecutionTool");  
hostProcLaunchTool.Name = "Emulator Host Process Execution Tool";
```

Create a `target.Command` object. Set the `String` property of the object to the command for starting the emulator.

```
emulatorCommand = target.create("Command", ...  
    String="pathToEmulatorApplication");
```

Specify the arguments of the command object as the configuration options for opening the emulator from command line.

```
emulatorCommand.Arguments = [ "-flag1","value1", ...  
    "-flag2","value2", ...  
    "-flagN","valueN"];
```

Specify the start command of the `target.HostProcessExecutionTool` object as the emulator command, and specify the launch tool of the emulator object as the host process execution tool.

```
hostProcLaunchTool.StartCommand = emulatorCommand;  
emulatorTarget.LaunchTool = hostProcLaunchTool;
```

Create a `target.TargetConnection` object to specify a connection between the host machine and the emulated target.

```
emulatorConnection = target.create( "TargetConnection", ...  
    Name = "Emulator Connection", ...  
    Target = emulatorBoard, ...  
    CommunicationType = "TCPChannel", ...  
    IPAddress = "localhost", ...  
    Port = "17725");
```

Make board, connection, and emulator objects persist across MATLAB sessions.

```
addedObjs = target.add([emulatorBoard , emulatorConnection , emulatorTarget]);
```

`target.add` summary:

```
Objects added to internal database for current MATLAB session:  
target.Board          "ARM Cortex-A Emulator-Based Test Board"  
target.Emulator        "Example Emulator Target"  
target.HostProcessExecutionTool  "Emulator Host Process Execution Tool"  
target.TCPChannel      "Emulator Connection TCPChannel"  
target.TargetConnection  "Emulator Connection"  
1 object not added because they already exist.
```


`emulatorBoard` is the `target.Board` object that describes the emulated target board.
`emulatorConnection` is the `target.TargetConnection` object that specifies the connection between Simulink and your target hardware, `emulatorTarget`.

To select the emulator for the model, set the `HardwareBoard` configuration parameter to the board name. For example:

```
set_param(model,HardwareBoard=board.Name);
```

You can now run PIL or external mode simulations on the emulator. For more information, see:

- “Set Up PIL Connectivity by Using Target Framework”
- “External Mode Simulation by Using XCP Communication”

To remove the added target objects, enter:

```
target.remove(addedObjs);
```

```
target.remove summary:
```

Objects removed from internal database:	
<code>target.Board</code>	"ARM Cortex-A Emulator-Based Test Board"
<code>target.Emulator</code>	"Example Emulator Target"
<code>target.HostProcessExecutionTool</code>	"Emulator Host Process Execution Tool"
<code>target.TCPChannel</code>	"Emulator Connection TCPChannel"
<code>target.TargetConnection</code>	"Emulator Connection"

Version History

Introduced in R2023b

See Also

`target.Board` | `target.HostProcessExecutionTool` | `target.create`

Topics

“Define Custom Emulator for Target Connectivity”

“Set Up PIL Connectivity by Using Target Framework”

“Set Up External Mode Connectivity Between Simulink and Target Hardware”

target.EnvironmentConfiguration Class

Namespace: target

Configure system environment for toolchain

Description

Use a `target.EnvironmentConfiguration` object to configure the system environment for a toolchain. For example, configuration of environment variables and execution of vendor-provided setup scripts.

To create a `target.EnvironmentConfiguration` object, use the `target.create` function.

Properties

HostOperatingSystemSupport — Operating system supported

`target.HostOperatingSystemSupport` object

Operating system that the environment configuration applies to.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

SetupCommand — Script or batch file with setup commands

`target.Command` object

Optional. Setup shell script or batch file that the software calls before executing CMake configure or build commands.

On a Windows computer, before invoking `cmake`, the software runs the `CALL` command from a batch file.

On a Linux[®] and Mac computers, before invoking `cmake`, the software runs the `source` or `.` command from a Bash shell script.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

SystemPaths — Paths to prepend to system path

string array

Optional. List of paths that must be prepended to the system path before executing `cmake` configure or build commands.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Examples

Configure System Environment for Toolchain

This code snippet from “Create Custom CMake Toolchain Definition” shows how you can use a `target.EnvironmentConfiguration` object to configure the system environment for a toolchain.

```
tc.EnvironmentConfiguration(1).HostOperatingSystemSupport.Linux = false;
tc.EnvironmentConfiguration(1).HostOperatingSystemSupport.Windows = true;
tc.EnvironmentConfiguration(1).HostOperatingSystemSupport.Mac = false;
tc.EnvironmentConfiguration(1).SystemPaths = ...
    {'$(MW_MINGW64_LOC)/bin', '$(MATLAB_ROOT)/toolbox/shared/coder/ninja/$(ARCH)'};

tc.EnvironmentConfiguration(2) = target.create('EnvironmentConfiguration');
tc.EnvironmentConfiguration(2).HostOperatingSystemSupport.Linux = true;
tc.EnvironmentConfiguration(2).HostOperatingSystemSupport.Windows = false;
tc.EnvironmentConfiguration(2).HostOperatingSystemSupport.Mac = true;
tc.EnvironmentConfiguration(2).SystemPaths = ...
    {'$(MATLAB_ROOT)/toolbox/shared/coder/ninja/$(ARCH)'};
```

Version History

Introduced in R2022b

See Also

`target.Toolchain` | `target.CMake` | `target.CMakeBuildType` | `target.CMakeBuilder` | `target.CMakeCacheEntry` | `target.HardwareComponentSupport` | `target.create`

Topics

“Create Custom CMake Toolchain Definition”

target.ExecutionService Class

Namespace: target

Describe implementation of execution service for target application

Description

Use the target.ExecutionService class, which inherits from target.ApplicationExecutionTool, to describe the execution service implementation for running an application on the target computer. The implementation uses MATLAB code.

To create a target.ExecutionService object, use the target.create function.

Properties

Name — Execution service name

character vector | string

The name of the target application execution service.

Attributes:

GetAccess	public
SetAccess	public

APIImplementation — API implementation

target.APIImplementation object

A target.APIImplementation object that describes the MATLAB data model service implementation.

Attributes:

GetAccess	public
SetAccess	public

HostOperatingSystemRequirements — Development computer operating system requirements

'All' (default) | 'Unix' | 'Linux' | 'MacOS' | 'Windows'

MathWorks system requirements for operating system of development computer.

Attributes:

GetAccess	public
SetAccess	public

Id — Object identifier

character vector | string

Value of the Name property.

Attributes:

GetAccess	public
SetAccess	private

Examples**Use target.ExecutionService to Describe Application Execution**

For an example that uses the `target.ExecutionService` class to describe the application execution within a debugger, see “Use Debugger for PIL Target Connectivity”.

Version History

Introduced in R2021a

See Also

`target.create` | `target.ExecutionTool` | `target.ApplicationExecutionTool`

Topics

“Use Debugger for PIL Target Connectivity”

target.ExecutionTool Class

Namespace: target

MATLAB service interface for tool that manages application execution on target hardware

Description

Use the `target.ExecutionTool` class to provide a MATLAB service interface for a tool that manages the application execution on the target hardware.

Properties

Application — Target application

string

Name of application to run on target hardware, which is set by MATLAB at runtime.

Attributes:

GetAccess	public
SetAccess	public

ApplicationCommandLineArguments — Command-line arguments

string array

Command-line arguments for the target application, which is set by MATLAB at runtime.

Attributes:

GetAccess	public
SetAccess	public

Connection — Connection details

target.TargetConnection object

Provide connection details.

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

<code>close</code>	Close target application harness
<code>getApplicationStatus</code>	Get status of target application
<code>getStandardError</code>	Return standard error stream from target application
<code>getStandardOutput</code>	Return standard output stream from target application
<code>loadApplication</code>	Load target application into harness
<code>open</code>	Open target application harness

startApplication	Start execution of target application
stopApplication	Stop execution of target application
unloadApplication	Unload target application from harness

Examples

Use target.ExecutionTool to Manage Application Execution

For examples that use the `target.ExecutionTool` class to manage application execution on the target hardware, see:

- “Set Up External Mode Connectivity Between Simulink and Target Hardware”. This workflow uses a subclass of `target.ExecutionTool`.
- “Use Debugger for PIL Target Connectivity”. This workflow uses `target.DebugIOTool`, which inherits from `target.ExecutionTool` through `target.DebugExecutionTool`.

.

Version History

Introduced in R2021a

See Also

`target.DebugExecutionTool` | `target.DebugIOTool`

Topics

“Use Debugger for PIL Target Connectivity”

close

Class: target.ExecutionTool

Namespace: target

Close target application harness

Syntax

```
errFlag = myExecutionTool.close()
```

Description

`errFlag = myExecutionTool.close()` closes the harness for the target application (that is managed by the associated execution tool) if it exists. An example of an application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

errFlag — Error flag

true | false

Outcome of the operation:

- `true` -- An error occurred while closing the application harness.
- `false` -- Method completed without error. Application harness closed or does not exist.

Version History

Introduced in R2021a

See Also

`target.ExecutionTool` | `open`

getApplicationStatus

Class: target.ExecutionTool

Namespace: target

Get status of target application

Syntax

```
[applicationStatus, errFlag] = myExecutionTool.getApplicationStatus()
```

Description

[applicationStatus, errFlag] = myExecutionTool.getApplicationStatus() returns the status of the target application (managed by the execution tool) and an outcome flag.

Output Arguments

applicationStatus — Target application status

target.ApplicationStatus member

The status of the target application that is being managed by the execution tool.

errFlag — Error flag

true | false

Outcome of the operation:

- `true` -- An error occurred while trying to obtain status of target application.
- `false` -- Status of target application obtained without error.

Version History

Introduced in R2021a

See Also

target.ExecutionTool | target.ApplicationStatus

getStandardError

Class: target.ExecutionTool

Namespace: target

Return standard error stream from target application

Syntax

```
[stdErrorStream, errFlag] = myExecutionTool.getStandardError()
```

Description

[stdErrorStream, errFlag] = myExecutionTool.getStandardError() returns, as a string, the standard error stream from the target application that is being managed by the associated execution tool. The method also returns an error flag.

Output Arguments

stdErrorStream — Standard error stream

string

The standard error stream from the beginning of the execution of the target application.

errFlag — Error flag

true | false

Outcome of the operation:

- `true` -- Error occurred while retrieving standard error stream from target application.
- `false` -- Standard error stream from target application returned.

Version History

Introduced in R2021a

See Also

target.ExecutionTool | getStandardOutput

getStandardOutput

Class: target.ExecutionTool

Namespace: target

Return standard output stream from target application

Syntax

```
[stdOutputStream, errFlag] = myExecutionTool.getStandardOutput()
```

Description

[stdOutputStream, errFlag] = myExecutionTool.getStandardOutput() returns, as a string, the standard output stream from the target application that is being managed by the associated execution tool. The method also returns an error flag.

Output Arguments

stdOutputStream — Standard output stream

string

The standard output stream from the beginning of the execution of the target application.

errFlag — Outcome flag

true | false

Outcome of the operation:

- **true** -- Error occurred while retrieving standard output stream from target application.
- **false** -- Standard output stream from target application returned.

Version History

Introduced in R2021a

See Also

target.ExecutionTool | getStandardError

loadApplication

Class: target.ExecutionTool

Namespace: target

Load target application into harness

Syntax

```
outcomeFlag = myExecutionTool.loadApplication()
```

Description

`outcomeFlag = myExecutionTool.loadApplication()` loads the target application (managed by the associated execution tool) into the application harness if the harness exists. An example of an application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

errFlag — Error flag

true | false

Outcome of the operation:

- `true` -- An error occurred while loading the target application into the harness.
- `false` -- Target application loaded into the harness or the harness does not exist.

Version History

Introduced in R2021a

See Also

`target.ExecutionTool` | `unloadApplication`

open

Class: target.ExecutionTool

Namespace: target

Open target application harness

Syntax

```
errFlag = myExecutionTool.open()
```

Description

`errFlag = myExecutionTool.open()` opens the harness for the target application (that is managed by the associated execution tool) if the harness exists. An example of a target application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

errFlag — Error flag

true | false

Outcome of the operation:

- true -- Error occurred while opening application harness.
- false -- Application harness opened or does not exist.

Version History

Introduced in R2021a

See Also

target.ExecutionTool | close

startApplication

Class: target.ExecutionTool

Namespace: target

Start execution of target application

Syntax

```
errFlag = myExecutionTool.startApplication()
```

Description

`errFlag = myExecutionTool.startApplication()` starts the execution of the target application (that is managed by the associated execution tool) and returns an error flag.

Output Arguments

errFlag — Error flag

true | false

Outcome of the operation:

- true -- Error occurred while trying to start target application.
- false -- Target application started.

Version History

Introduced in R2021a

See Also

target.ExecutionTool | stopApplication

stopApplication

Class: target.ExecutionTool

Namespace: target

Stop execution of target application

Syntax

```
errFlag = myExecutionTool.stopApplication()
```

Description

`errFlag = myExecutionTool.stopApplication()` stops the execution of the target application (that is managed by the associated execution tool) and returns an error flag.

Output Arguments

errFlag — Error flag

true | false

Outcome of the operation:

- true -- Error occurred while trying to stop target application.
- false -- Target application stopped.

Version History

Introduced in R2021a

See Also

target.ExecutionTool | startApplication

unloadApplication

Class: target.ExecutionTool

Namespace: target

Unload target application from harness

Syntax

```
outcomeFlag = myExecutionTool.unloadApplication()
```

Description

`outcomeFlag = myExecutionTool.unloadApplication()` unloads the target application (managed by the associated execution tool) from the application harness if the harness exists. An example of an application harness is a simulator or debugger. The method returns an error flag.

Output Arguments

errFlag — Error flag

true | false

Outcome of the operation:

- `true` -- An error occurred while unloading the target application from the harness.
- `false` -- Target application unloaded from the harness or the harness does not exist.

Version History

Introduced in R2021a

See Also

`target.ExecutionTool` | `loadApplication`

target.export

Namespace: target

Export target object data

Syntax

```
target.export(targetObject)
target.export(targetObject,Name,Value)
```

Description

`target.export(targetObject)` exports the specified target object data to a MATLAB function, `registerTargets.m`. Use the generated file to share target feature data between sessions and computers. When you run `registerTargets.m`, it recreates the target object and adds the object to an internal database.

`target.export(targetObject,Name,Value)` exports the specified target object data using one or more name-value pair arguments.

Examples

Share Hardware Device Data

You can share hardware device data across computers and users.

Specify two hardware devices.

```
langImp1 = target.create('LanguageImplementation', ...
                        'Name', 'MyLanguageImplementation1', ...
                        'Copy', 'ARM Compatible-ARM Cortex');

langImp2 = target.create('LanguageImplementation', ...
                        'Name', 'MyLanguageImplementation2', ...
                        'Copy', 'Atmel-AVR');

myProc1 = target.create('Processor','Name','MyProcessor1');
myProc1.LanguageImplementations = [langImp1, langImp2];
objectsAdded1 = target.add(myProc1, ...
                          'UserInstall',true, ...
                          'SuppressOutput',true);

myProc2 = target.create('Processor','Name','MyProcessor2');
objectsAdded2 = target.add(myProc2, ...
                          'UserInstall',true, ...
                          'SuppressOutput',true);
```

Run the `target.export` function.

```
target.export([myProc1, myProc2], 'FileName', 'exportMyProcFunction')
```

The function generates `exportMyProcFunction.m` in the current working folder. Use the generated function to share hardware device data across computers and users. For example, on another computer, run this command:

```
addedObjects = exportMyProcFunction;
```

The generated function recreates and adds the objects to an internal database.

If you want the hardware device data to persist over MATLAB sessions, run:

```
addedObjects = exportMyProcFunction('UserInstall',true);
```

Input Arguments

targetObject — Target object

object vector

Specify the target objects that you want to export.

Example: `target.export(myTargetObject);`

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: `target.export(myTargetObject, 'FileName', 'exportMyTargetFn');`

FileName — File name

character vector | string

Specify name of MATLAB function file that contains exported target data.

Overwrite — Overwrite flag

false (default) | true

- `true` -- Overwrite MATLAB function file if it exists.
- `false` -- Generate error if MATLAB function file exists. File is not overwritten.

Version History

Introduced in R2019b

See Also

`target.add` | `target.create`

Topics

“Register New Hardware Devices”

target.ExternalMode Class

Namespace: target

Represent external mode protocol stack

Description

Use the target.ExternalMode class, which is a subclass of target.CommunicationProtocolStack, to specify the external mode protocol stack for your target hardware.

To create a target.ExternalMode object, use the target.create function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

Properties

Connectivities — External mode connectivity options

target.ExternalModeConnectivity object array

Provide connectivity options for the external mode protocol stack. The array can contain only one target.ExternalModeConnectivity object for a specific transport protocol. For example, the array can contain one object for XCP on TCP/IP and another object for XCP on Serial.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify External Mode Protocol Stack for Target Hardware

This code snippet from “Customize Connectivity for XCP External Mode Simulations” shows how to specify the external mode protocol stack for your target hardware.

```
xcpPlatformAbstraction = target.create('XCPPlatformAbstraction', ...
    'Name', 'XCP Platform Abstraction');

xcpPlatformAbstraction.BuildDependencies.Defines = {'XCP_CUSTOM_PLATFORM'};
customPlatformAbstractionPath = 'pathToImplementationFolder';
xcpPlatformAbstraction.BuildDependencies.SourceFiles = ...
    {fullfile(customPlatformAbstractionPath, 'myXCPPlatform.c')};
xcpPlatformAbstraction.BuildDependencies.IncludePaths = ...
    {customPlatformAbstractionPath};

xcpTransport = target.create('XCPTCPIPTransport', ...
    'Name', 'XCP Transport');

xcpConfiguration = target.create('XCP', ...
    'Name', 'XCP Configuration', ...
    'XCPTTransport', xcpTransport, ...
    'XCPPlatformAbstraction', xcpPlatformAbstraction);
```

```
extModeTCPConnectivity = ...
    target.create('XCPEExternalModeConnectivity', ...
        'Name', 'External Mode TCP Connectivity', ...
        'XCP', xcpConfiguration);

externalMode = target.create('ExternalMode', ...
    'Name', 'External Mode', ...
    'Connectivities', extModeTCPConnectivity);

board.CommunicationProtocolStacks = externalMode;
```

Note You can create the `target.ExternalMode` object in a single step.

```
externalMode = target.create('ExternalMode', ...
    'Name', 'External Mode', ...
    'XCPTransportLayer', 'TCP', ...
    'Defines', {'XCP_CUSTOM_PLATFORM'}, ...
    'SourceFiles', {fullfile('pathToImplementationFolder', 'myXCPPlatform.c')}, ...
    'IncludePaths', {'pathToImplementationFolder'});
```

Version History

Introduced in R2021a

See Also

`target.create` | `target.Board` | `target.CommunicationProtocolStack` | `target.XCP` | `target.ExternalModeConnectivity` | `target.XCPEExternalModeConnectivity`

Topics

“Customize Connectivity for XCP External Mode Simulations”

target.ExternalModeConnectivity Class

Namespace: target

Base class for external mode connectivity options

Description

The target.ExternalModeConnectivity class is a base class for representing connectivity options that are available in the external mode protocol stack. The target.XCPExternalModeConnectivity class is derived from this base class.

Class Attributes

Abstract	true
HandleCompatible	true

For information on class attributes, see “Class Attributes”.

Version History

Introduced in R2021a

See Also

target.create | target.Board | target.CommunicationProtocolStack | target.ExternalMode | target.XCPExternalModeConnectivity

Topics

“Customize Connectivity for XCP External Mode Simulations”

target.FileType Class

Namespace: target

Define identifier for file type

Description

The software provides `target.FileType` objects that define identifiers for different file types.

You cannot create `target.FileType` objects.

Properties

Id — Object identifier

string

Unique ID of object in internal database.

Attributes:

GetAccess	public
SetAccess	private

Name — File type name

string

Name of file type. The software provides objects with these names:

- Source
- Object
- Static Library
- Shared Library
- Executable
- Dependency
- HEX File
- BIN File
- Debug Symbols
- Makefile
- Header

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2023a

See Also

[target.BuildTool](#) | [target.BuildToolType](#) | [target.Directive](#) |
[target.MakefileBuilder](#) | [target.MakeToolType](#) | [target.PairedDirective](#) |
[target.RepeatingDirective](#) | [target.Toolchain](#)

Topics

[“Define Custom Makefile-Based Toolchains Using Target Framework”](#)

target.FloatingPointDataType Class

Namespace: target

Describe floating point data type implemented by compiler for target hardware

Description

Use a `target.FloatingPointDataType` object to describe a floating point data type that is implemented by the target hardware compiler but is not specified by a C language standard.

To create a `target.FloatingPointDataType` object, use the `target.create` function.

The `target.FloatingPointDataType` class is a handle class.

Properties

Size — Size in bits

0 (default) | integer

Size of the data type, specified in bits.

Attributes:

GetAccess	public
SetAccess	public

TypeName — Data type name

character vector | string scalar

Name of floating point data type.

Attributes:

GetAccess	public
SetAccess	public

LiteralSuffix — Numeric literal suffix

character vector | string scalar

Describe the suffix appended to numeric literal in C or C++ code, which indicates data type. For example, some ARM® compilers support the use of the literal suffix `f16` for the `_Float16` data type, which allows the specification of literals of the data type in C or C++ code, say, `1.23f16`.

Attributes:

GetAccess	public
SetAccess	public

Includes — Required C/C++ header files

cell of character vectors | string array

For the data type, specify the include files that are required in the C or C++ translation unit.

Attributes:

GetAccess	public
SetAccess	public

SystemIncludes — Required C/C++ system header files

cell of character vectors | string array

For the data type, specify the system include files that are required in the C or C++ translation unit.

Attributes:

GetAccess	public
SetAccess	public

Name — Identifier

string scalar

Identifier for data type description.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify Half-Precision ARM Data Type

Specify a 16-bit floating point ARM data type.

```
languageImplementation = target.create('LanguageImplementation', ...
    'Name', 'ARM with half', ...
    'Copy', 'ARM Compatible-ARM Cortex');

customHalf = target.create('FloatingPointDataType', ...
    'Name', 'ARM Half Float', ...
    'TypeName', '_Float16', ...
    'LiteralSuffix', 'f16', ...
    'Size', 16, ...
    'SystemIncludes', "arm_fp16.h");
languageImplementation.DataTypes.NonStandardDataTypes = [customHalf];
target.add(languageImplementation);
```

Version History

Introduced in R2021b

See Also

target.create | target.add | target.remove

target.Function Class

Namespace: target

Provide function signature information

Description

Use the `target.Function` class, which inherits functionality from `target.Data`, to provide function signature information.

To create a `target.Function` object, use the `target.create` function.

Properties

ReturnType — Data type returned

character vector | string

Data type of value that the associated function returns.

Attributes:

GetAccess	public
SetAccess	public

Name — Function name

character vector | string

Name of function.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create Function Signature

Create the function signature for a timer by using the `target.Function` class.

Create the signature for a function that returns the data type `uint64` and the function name `timestamp_x86`.

```
timerSignature = target.create('Function');  
timerSignature.Name = 'timestamp_x86';  
timerSignature.ReturnType = 'uint64';
```

Version History

Introduced in R2020b

See Also

`target.create`

target.get

Namespace: target

Retrieve target objects from internal database

Syntax

```
targetObject = target.get(targetType, targetObjectId)
tFOList = target.get(targetType)
tFOList = target.get(targetType, Name, Value)
```

Description

`targetObject = target.get(targetType, targetObjectId)` retrieves a target object from an internal database.

`tFOList = target.get(targetType)` returns a list of *targetType* objects that are stored in the internal database.

`tFOList = target.get(targetType, Name, Value)` returns a list of *targetType* objects that have properties that match the name-value pairs.

Examples

Remove Target Object

This example shows how you can remove a `target.LanguageImplementation` object associated with an object identifier, *myLanguageImplementationID*.

Retrieve the object from the internal database.

```
objectToRemove = target.get('LanguageImplementation', myLanguageImplementationID);
```

Remove the object.

```
target.remove(objectToRemove);
```

Create Board Description

This example shows how to create a `target.Board` object that provides MATLAB with a description of processor attributes. It uses `target.get` to retrieve the description of a supported processor.

Create a board object.

```
hostTarget = target.create('Board', 'Name', 'Host Intel processor');
```

Specify the processor for the board by reusing a supported processor.

```
hostTarget.Processors = target.get('Processor', ...
                                   'Intel-x86-64 (Linux 64)');
```

Input Arguments

targetType — Target type

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'`.

For the list of supported classes, see `target`.

targetObjectId — Target 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

targetObject — Target object

object

Retrieved target object. For example:

- If `targetType` is `'Processor'`, the returned object is a `target.Processor` object.
- If `targetType` is `'LanguageImplementation'`, the returned object is a `target.LanguageImplementation` object.

For the list of supported classes, see `target`.

tF0List — Target object list

object vector

List of retrieved target objects.

Version History

Introduced in R2019a

See Also

`target.add` | `target.create` | `target.remove`

Topics

“Register New Hardware Devices”

target.HardwareComponentSupport Class

Namespace: target

Describe support for a hardware component

Description

Use a `target.HardwareComponentSupport` object to specify that a tool or toolchain applies to a particular target device.

To create a `target.HardwareComponentSupport` object, use the `target.create` function.

The `target.HardwareComponentSupport` class is a handle class.

Properties

Component — Supported target device

target object

Supported target device, for example, a `target.Processor` or a `target.Board` object.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Examples

Use target.HardwareComponentSupport to Specify Toolchain Support for Target Device

This code snippet shows how you can create a `target.HardwareComponentSupport` object and use it to define a toolchain that is available only when the target device Intel-x86-64 (Windows64) is selected.

```
tc = target.create('Toolchain', 'Name', 'My Toolchain');
tc.Builder = target.create('CMakeBuilder', 'Generator', 'Ninja');
tc.SupportedHardware = target.create('HardwareComponentSupport');
tc.SupportedHardware.Component = target.get('Processor', 'Intel-x86-64 (Windows64)');
```

Version History

Introduced in R2022b

See Also

`target.Toolchain` | `target.CMake` | `target.CMakeBuildType` | `target.CMakeBuilder` | `target.CMakeCacheEntry` | `target.EnvironmentConfiguration` | `target.create`

Topics

“Create Custom CMake Toolchain Definition”

target.HostProcessExecutionTool Class

Namespace: target

Capture system command information to run target application from MATLAB computer

Description

Use the `target.HostProcessExecutionTool` class to capture system command information that is required to run the target application from your development computer. Use this class to specify nonblocking commands. MATLAB runs a nonblocking command in the background and does not wait for the termination of the command.

Properties

Name — Execution tool name

character vector | string

Name of the execution tool.

Attributes:

GetAccess	public
SetAccess	public

StartCommand — Command list to run application

target.Command object

A `target.Command` object that provides a system command for running the application. The command in the list starts the application process.

This property must not be empty.

Attributes:

GetAccess	public
SetAccess	public

StopCommand — Command list to stop application

target.Command object

A `target.Command` object that provides a system command to stop the application execution.

Attributes:

GetAccess	public
SetAccess	public

RunsOn — Context to run tool

empty `target.ExecutionContext` array (default) | `target.SSHExecutionContext` object

Context to run the execution tool, specified as an empty `target.ExecutionContext` array (default) or a `target.SSHExecutionContext` object.

Context to Run Tool	Property Value	Specified By Example
Over SSH Connection	target.SSHExecutionContext object	executionTool.RunsOn = target.ExecutionContext
Locally (default)	Empty target.ExecutionContext array	executionTool.RunsOn = target.ExecutionContext

Attributes:

GetAccess

SetAccess

public

public

Id — Object identifier

character vector | string

Value of the Name property.

Attributes:

GetAccess

SetAccess

public

private

Version History

Introduced in R2020b

See Also

target.create | target.ApplicationExecutionTool | target.Command | target.SystemCommandExecutionTool

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.LanguageImplementation Class

Namespace: 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

Endianness — Byte ordering

'Little' (default) | 'Big' | 'Unspecified'

Byte ordering implemented by target hardware.

Attributes:

GetAccess	public
SetAccess	public

DataTypes — Data type definitions

target.DataTypes object

Provides C and C++ data type definitions through properties described in this table.

Property Name	Purpose
Char	target.DataTypes.Char object for char.
Short	target.DataTypes.Short object for short.
Int	target.DataTypes.Int object for int.
Long	target.DataTypes.Long object for long.
LongLong	target.DataTypes.LongLong object for longlong.
Half	target.DataTypes.Half object for a half precision data type that the target hardware uses.
Float	target.DataTypes.Float object for float.
Double	target.DataTypes.Double object for double.
Pointer	target.DataTypes.Pointer object for pointer.
SizeT	target.DataTypes.SizeT object for size_t.
PtrDiffT	target.DataTypes.PtrDiffT object for ptrdiff_t.
NonStandardDataTypes	target.FloatingPointDataType object array for target hardware data types that are not specified by C language standards.

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**Specify Hardware Implementation for New Device**

This example shows how to register a new hardware device.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create("Processor", Name="MyProcessor", ...
    Manufacturer="MyManufacturer");
```

Create a `target.LanguageImplementation` object for language implementation details.

```
myLanguageImplementation = target.create("LanguageImplementation", ...
    Name="MyProcessorImplementation");
```

Specify language implementation details.

```
myLanguageImplementation.Endianness = target.Endianness.Little;
```

```
myLanguageImplementation.AtomicIntegerSize = 64;
myLanguageImplementation.AtomicFloatSize = 64;
myLanguageImplementation.WordSize = 64;
```

```
myLanguageImplementation.DataTypes.Char.Size = 8;
myLanguageImplementation.DataTypes.Short.Size = 16;
myLanguageImplementation.DataTypes.Int.Size = 32;
myLanguageImplementation.DataTypes.Long.Size = 64;
myLanguageImplementation.DataTypes.LongLong.IsSupported = true;
myLanguageImplementation.DataTypes.LongLong.Size = 64;
myLanguageImplementation.DataTypes.Float.Size = 32;
myLanguageImplementation.DataTypes.Double.Size = 64;
```

```
myLanguageImplementation.DataTypes.Pointer.Size = 32;
```

```
myLanguageImplementation.DataTypes.SizeT.Size = 64;
myLanguageImplementation.DataTypes.PtrDiffT.Size = 64;
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myLanguageImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

```
target.add summary:
```

```
    Objects added to internal database for current MATLAB session:
```

```
target.LanguageImplementation    "MyProcessorImplementation"  
target.Processor               "MyManufacturer-MyProcessor"
```

- If you are using the MATLAB® Coder™: On the **Hardware** tab, you see the new device. Alternatively, you can now create a `coder.Hardware` object for this device by using the `coder.hardware` function.
- If you are using the Simulink® Coder™: On the **Hardware Implementation** pane, you can now set **Device vendor** and **Device type** to `MyManufacturer` and `MyProcessor` respectively.

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded)
```

```
target.remove summary:
```

```
Objects removed from internal database:  
target.LanguageImplementation    "MyProcessorImplementation"  
target.Processor               "MyManufacturer-MyProcessor"
```

Create Hardware Implementation by Modifying Existing Implementation

If an existing hardware implementation contains most of the values that you want in a new hardware implementation, you can quickly create the new implementation by creating and modifying a copy of the existing implementation.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create("Processor",Name="MyProcessor", ...  
                      Manufacturer="MyManufacturer");
```

Create a `target.LanguageImplementation` object that copies an existing language implementation.

```
myCopiedImplementation = target.create("LanguageImplementation", ...  
                                     Name="MyCopiedImplementation", ...  
                                     Copy="Atmel-AVR");
```

Specify the required language implementation details. For example, byte ordering.

```
myCopiedImplementation.Endianess = target.Endianess.Big;
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myCopiedImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

```
target.add summary:
```

```
Objects added to internal database for current MATLAB session:  
target.LanguageImplementation    "MyCopiedImplementation"  
target.Processor               "MyManufacturer-MyProcessor"
```

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded)
```

```
target.remove summary:
```

```
Objects removed from internal database:
target.LanguageImplementation    "MyCopiedImplementation"
target.Processor                "MyManufacturer-MyProcessor"
```

Create Hardware Implementation by Reusing Existing Implementation

If your hardware device requires the same hardware implementation as an existing implementation, you can reuse the existing implementation.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create( "Processor", Name="MyProcessor", ...
                        Manufacturer="MyManufacturer");
```

Retrieve the existing implementation by using the identifier for the device vendor and type.

```
existingImplementation = target.get("LanguageImplementation", ...
                                   "ARM Compatible-ARM Cortex");
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = existingImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

```
target.add summary:
```

```
Objects added to internal database for current MATLAB session:
target.Processor                "MyManufacturer-MyProcessor"
1 object not added because they already exist.
```

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded);
```

```
target.remove summary:
```

```
Objects removed from internal database:
target.Processor    "MyManufacturer-MyProcessor"
```

Version History

Introduced in R2019a

See Also

`target.Processor` | `target.create`

Topics

“Register New Hardware Devices”

target.MainFunction Class

Namespace: target

Provide C and C++ dependencies for main function of target hardware application

Description

Use the `target.MainFunction` class to provide main function dependencies for an application main function that runs on your target hardware. For example, C and C++ initialization and termination code, include preprocessor directives, and specification of main function arguments for the application.

To create a `target.MainFunction` object, use the `target.create` function.

Properties

Name — Dependency collection

character vector | string

Name of the collection of main dependencies.

Attributes:

GetAccess	public
SetAccess	public

Dependencies — Build dependencies

`target.BuildDependencies` object

Compiler build tool dependencies of the main function, which include header files, source files, and libraries.

Attributes:

GetAccess	public
SetAccess	public

Arguments — Command-line arguments

string array

Capture run-time command-line argument dependencies.

Attributes:

GetAccess	public
SetAccess	public

IncludeFiles — #include files

string array

Array of header files that must be included in a target main function by using the preprocessor directive `#include "path-spec"`.

Attributes:

GetAccess	public
SetAccess	public

SystemIncludeFiles — System #include files

string array

Array of header files that must be included in a target main function by using the preprocessor directive `#include <path-spec>`.

Attributes:

GetAccess	public
SetAccess	protected

InitializationCode — Target main initialization

character vector | string

Formatted string of C or C++ code that the main function uses to initialize target resources.

Attributes:

GetAccess	public
SetAccess	public

TerminationCode — Target main termination

character vector | string

Formatted string of C or C++ code that the main function uses to terminate target resources.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify Target-Specific main Function Dependencies

Create a `target.MainFunction` object and associate it with a `target.Board` object, which captures the main function dependencies for an Arduino board. Workflows, such as processor-in-the-loop (PIL), can use this information when generating a main function for an application that runs on the target hardware.

```
board = target.create('Board', 'Name', 'Arduino Board')
mainFunction = target.create('MainFunction');
mainFunction.Name = 'Arduino Main Dependencies';

mainFunction.IncludeFiles = { 'Arduino.h' };
mainFunction.InitializationCode = fileread('arduino_main_initialization.c');

board.MainFunctions = mainFunction;
```

In the code snippet, `arduino_main_initialization.c` contains C code. For example:


```
/* Initialize system */
init();
```

Specify main Function Run-Time Arguments

This code snippet from “Set Up PIL Connectivity by Using Target Framework” shows how you can create and use a `target.MainFunction` object to specify main function arguments that are required for an API implementation.

```
comms = target.create('CommunicationInterface');
comms.Name = 'Linux TCP Interface';
comms.Channel = 'TCPChannel';
comms.APIImplementations = target.create('APIImplementation', ...
    'Name', 'x86_rtiostream Implementation');
comms.APIImplementations.API = target.create('API', 'Name', 'rtiostream');
comms.APIImplementations.BuildDependencies = target.create('BuildDependencies');
comms.APIImplementations.BuildDependencies.SourceFiles = ...
    {fullfile('${MATLABROOT}', ...
        'toolbox', ...
        'coder', ...
        'rtiostream', ...
        'src', ...
        'rtiostreamtcpip', ...
        'rtiostream_tcpip.c')};
comms.APIImplementations.MainFunction = target.create('MainFunction', ...
    'Name', 'TCP RtIOStream Main');
comms.APIImplementations.MainFunction.Arguments = {'-blocking', '1', '-port', '0'};
hostTarget.CommunicationInterfaces = comms;
```

Version History

Introduced in R2020b

See Also

`target.create` | `target.Board` | `target.APIImplementation`

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.MakefileBuilder Class

Namespace: target

Specify that toolchain is makefile-based

Description

Use a `target.MakefileBuilder` object to:

- Specify that the toolchain is makefile-based.
- Specify the type of makefile that the toolchain generates (GNU® Make or NMAKE).

To create a `target.MakefileBuilder` object, use the `target.create` function. Create the object in a single step.

```
makefileBuilderObject = target.create('MakefileBuilder', option)
```

option is the name of the `target.MakeToolType` object that you want to specify for the `MakeToolType` property of *makefileBuilderObject*. For example, 'GMake' or 'NMake'.

Properties

MakeToolType — Makefile type

`target.MakeToolType` object

Type of makefile that the toolchain generates. Specify either GMake or NMake.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify Makefile-Based Toolchain

This code snippet shows how you can use a `target.MakefileBuilder` object to specify that the toolchain is based on a GNU Make (gmake) makefile.

```
mingwtec.Builder = target.create('MakefileBuilder', 'GMake');
```

Version History

Introduced in R2023a

See Also

`target.BuildTool` | `target.BuildToolType` | `target.Directive` | `target.FileType` | `target.MakeToolType` | `target.PairedDirective` | `target.RepeatingDirective` | `target.Toolchain`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.MakeToolType Class

Namespace: target

Describe syntax for makefile type

Description

The software provides `target.MakeToolType` objects that describe the syntax for specific makefile types.

You cannot create `target.MakeToolType` objects.

Properties

Id — Object identifier

string

Unique ID of object in internal database.

Attributes:

GetAccess	public
SetAccess	private

Name — Makefile type

string

Name of makefile type. The software provides objects with these names:

- GMake
- NMake

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2023a

See Also

`target.BuildTool` | `target.BuildToolType` | `target.Directive` | `target.FileType` | `target.MakefileBuilder` | `target.PairedDirective` | `target.RepeatingDirective` | `target.Toolchain`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.MATLABDependencies Class

Namespace: target

Describe MATLAB class and function dependencies

Description

Use the `target.MATLABDependencies` class to describe MATLAB class and function dependencies.

To create a `target.MATLABDependencies` object, use the `target.create` function.

Properties

Classes — MATLAB classes

string array

MATLAB classes that make up the implementation dependencies.

Attributes:

GetAccess	public
SetAccess	public

Functions — MATLAB functions

string array

MATLAB functions that make up the implementation dependencies.

Attributes:

GetAccess	public
SetAccess	public

Examples

MATLAB Dependencies of `target.DebugIOTool` Implementation

For an example that describes the MATLAB dependencies of an implementation of `target.DebugIOTool`, see “Use Debugger for PIL Target Connectivity”.

Version History

Introduced in R2021a

See Also

`target.create` | `target.API` | `target.APIImplementation` | `target.BuildDependencies`

Topics

“Use Debugger for PIL Target Connectivity”

target.Object Class

Namespace: target

Base class for target types

Description

`target.Object` is an abstract base class that enables target types to inherit common functionality.

Class Attributes

Abstract	true
HandleCompatible	true

For information on class attributes, see “Class Attributes”.

Properties

IsValid — Data validity

true | false

- true if object passes validation procedure.
- false otherwise.

Attributes:

GetAccess	public
SetAccess	private

Data Types: logical

Methods

Public Methods

`validate` Validate data integrity of target feature object

Examples

Validate Hardware Device Data

To validate the data integrity of target objects, use the `IsValid` property or the `validate` method of the `target.Object` base class.

Consider an example where you create a `target.Processor` object and associate an existing language implementation with the object.

```
myProcessor = target.create("Processor");  
myProcessor.LanguageImplementations = target.get("LanguageImplementation", ...  
                                                "ARM Compatible-ARM Cortex");
```

To see that the newly created object is not valid, enter `myProcessor.IsValid`.

```
myProcessor.IsValid
```

```
ans = logical
      0
```

If you try to validate the object with the method `myProcessor.validate()`, you get an error.

```
myProcessor.validate()
```

```
Error using target.internal.Processor/validate
Target data validation failed.
* Undefined property "Name" in "Processor" object.
* Undefined identifier in "Processor" object.
```

The validation fails because these `target.Processor` properties are not specified:

- Name — Processor name
- Id — Object identifier

You can specify a processor name, which also specifies the object identifier.

```
myProcessor.Name = "MyProcessor";
```

Check the validity of `myProcessor` again to see that the validity of the object is established.

```
myProcessor.IsValid
```

```
ans = logical
      1
```

```
myProcessor.validate()
```

Note: When you use the `target.add` function to register a target object, the software also checks the validity of the object.

Version History

Introduced in R2019b

See Also

Topics

“Register New Hardware Devices”

validate

Class: target.Object

Namespace: target

Validate data integrity of target feature object

Syntax

```
myTargetFeature.validate()
```

Description

`myTargetFeature.validate()` runs a procedure to validate the data integrity of the object *myTargetFeature*. If the validation procedure fails, the method produces an error.

Examples

Validate Hardware Device Data

To validate the data integrity of target objects, use the `IsValid` property or the `validate` method of the `target.Object` base class.

Consider an example where you create a `target.Processor` object and associate an existing language implementation with the object.

```
myProcessor = target.create("Processor");  
myProcessor.LanguageImplementations = target.get("LanguageImplementation", ...  
                                                "ARM Compatible-ARM Cortex");
```

To see that the newly created object is not valid, enter `myProcessor.IsValid`.

```
myProcessor.IsValid
```

```
ans = logical  
      0
```

If you try to validate the object with the method `myProcessor.validate()`, you get an error.

```
myProcessor.validate()
```

```
Error using target.internal.Processor/validate  
Target data validation failed.  
* Undefined property "Name" in "Processor" object.  
* Undefined identifier in "Processor" object.
```

The validation fails because these `target.Processor` properties are not specified:

- Name — Processor name
- Id — Object identifier

You can specify a processor name, which also specifies the object identifier.

```
myProcessor.Name = "MyProcessor";
```

Check the validity of `myProcessor` again to see that the validity of the object is established.

```
myProcessor.IsValid
```

```
ans = logical  
    1
```

```
myProcessor.validate()
```

Note: When you use the `target.add` function to register a target object, the software also checks the validity of the object.

Version History

Introduced in R2019b

See Also

Topics

“Register New Hardware Devices”

target.OperatingSystem Class

Namespace: target

Superclasses: target.Object

Describe operating system for target hardware

Description

Use a target.OperatingSystem object to describe the operating system for your target hardware.

Class Attributes

HandleCompatible true

For information on class attributes, see “Class Attributes”.

Creation

- Create a target.OperatingSystem object by using the target.create function.
- Retrieve an existing target.OperatingSystem object by using the target.get function.

Properties

Name — Object name

string scalar | character vector

Object name, specified as a string scalar or character vector.

Attributes:

GetAccess	public
SetAccess	public

Data Types: string | char

Families — Operating system families

string scalar | character vector

Operating system families to which the operating system represented by this object belongs, specified as a string scalar or character vector. Each family is a set of operating systems that are compatible with a specific feature.

Attributes:

GetAccess	public
SetAccess	public

Examples

Use of target.OperatingSystem Object

For a workflow that uses a `target.OperatingSystem` object, see “Set Up PIL Connectivity for Linux-Based Target Hardware”.

Version History

Introduced in R2026a

See Also

`target.create` | `target.Processor` | `target.Board` | `target.Support`

Topics

“Set Up PIL Connectivity for Linux-Based Target Hardware”

target.PairedDirective Class

Namespace: target

Describe pair of command-line flags

Description

Use a `target.PairedDirective` object to describe a pair of command-line flags that you use together. For example, `--start-group` and `--end-group`, which enables multi-pass linking of a group of libraries.

To create a `target.PairedDirective` object, use the `setDirective` method of the `target.BuildTool` object.

Properties

Name — Directive pair name

string

Name of directive pair.

Attributes:

GetAccess	public
SetAccess	public

StartValue — Start directive value

string

Value of start directive.

Attributes:

GetAccess	public
SetAccess	public

EndValue — End directive value

string

Value of end directive.

Attributes:

GetAccess	public
SetAccess	public

Examples

Specify Linker that Groups Libraries

This code snippet shows how you can specify a C linker that groups libraries that have circular dependencies.

```
cLinker = target.create('BuildTool', 'Linker', 'gcc', ...  
    'Name', 'MinGW Linker', ...  
    'HostOperatingSystemSupport', target.HostOperatingSystemSupport.WindowsOnly);  
cLinker.setDirective('LibraryGroup', '-WL,--start-group', '-WL,--end-group');
```

Version History

Introduced in R2023a

See Also

target.BuildTool | target.BuildToolType | target.Directive | target.FileType |
target.MakefileBuilder | target.MakeToolType | target.RepeatingDirective |
target.Toolchain

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.PILProtocol Class

Namespace: target

Describe PIL protocol implementation for target hardware

Description

Use the `target.PILProtocol` class, which inherits functionality from `target.CommunicationProtocolStack`, to describe the processor-in-the-loop (PIL) communication protocol implementation for your target hardware. For example, use this class to provide buffering information for data transfer and timeout information for I/O with the associated `target.Board` object.

To create a `target.PILProtocol` object, use the `target.create` function.

Properties

Name — PIL protocol object name

character vector | string

Name of the PIL protocol object.

Attributes:

GetAccess	public
SetAccess	public

SendBufferSize — Send buffer size

scalar integer

Size of send buffer for caching communication data.

Attributes:

GetAccess	public
SetAccess	public

ReceiveBufferSize — Receive buffer size

scalar integer

Size of receive buffer for caching communication data.

Attributes:

GetAccess	public
SetAccess	public

SendTimeout — Send timeout

scalar integer

Timeout that is applied to a data send command, specified in seconds.

Attributes:

GetAccess	public
SetAccess	public

ReceiveTimeout — Receive timeout

scalar integer

Timeout that is applied to a data receive command, specified in seconds.

Attributes:

GetAccess	public
SetAccess	public

OpenTimeout — Open timeout

scalar integer

Timeout that is applied when opening PIL communications, specified in seconds.

Attributes:

GetAccess	public
SetAccess	public

ByteInputOutputOnly — Bytes only

scalar boolean

Specify whether PIL communication sends and receives only bytes.

Attributes:

GetAccess	public
SetAccess	public

Examples**Specify PIL Protocol Information**

Specify PIL protocol information. This code snippet from “Set Up PIL Connectivity by Using Target Framework” shows how to specify the information.

```

pilProtocol = target.create('PILProtocol');
pilProtocol.Name = 'Linux PIL Protocol';
pilProtocol.SendBufferSize = 50000;
pilProtocol.ReceiveBufferSize = 50000;
hostTarget.CommunicationProtocolStacks = pilProtocol;

```

Version History

Introduced in R2020b

See Also
[target.create](#) | [target.CommunicationProtocolStack](#)

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.Port Class

Namespace: target

Describe connection via target hardware port

Description

Use the `target.Port` class, which inherits from `target.ConnectionProperties`, to describe a connection via a port of the target hardware. For example, the serial COM port.

Properties

AddOns — Custom property add-on objects

`target.AddOns` object array

To provide additional custom properties, associate one or more `target.AddOn` objects with the `target.Port` object.

Attributes:

GetAccess	public
SetAccess	public

PortNumber — Port number

`string`

Number of the port associated with the connection.

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

get	Get name of custom add-on property
set	Set property value

Version History

Introduced in R2021a

See Also

`target.create` | `target.ConnectionProperties`

target.PortConnection Class

Namespace: target

Describe target connection port

Description

Use the `target.PortConnection` class, which inherits functionality from `target.ConnectionProperties`, to describe a target connection port. For example, a serial COM port.

To create a `target.PortConnection` object, use the `target.create` function.

Properties

Port — Port

character vector | string

Port number.

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2020b

See Also

`target.create` | `target.ConnectionProperties`

target.Processor Class

Namespace: 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:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>private</code>

LanguageImplementations — Language implementation

object

Associated `target.LanguageImplementation` object.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Name — Processor name

character vector | string

Name of the target processor.

Example: `'Cortex-A53'`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Manufacturer — Processor manufacturer

character vector | string

Optional description of the target processor manufacturer.

Example: `'ARM Compatible'`

Attributes:

GetAccess	public
SetAccess	public

Timers — Timers

`target.Counter` object array

Provide timer information.

Attributes:

GetAccess	public
SetAccess	public

Overheads — Profiling instrumentation overheads

vector

Specify instrumentation overhead values for removal from execution-time measurements.

Attributes:

GetAccess	public
SetAccess	public

NumberOfCores — Processor cores

1 (default) | scalar

Number of physical cores in processor.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `uint`

NumberOfThreadsPerCore — Threads per processor core

1 (default) | scalar

Number of threads per processor core.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `uint`

NumberOfLogicalCores — Logical cores

scalar

Number of logical cores that processor provides, which is equal to `NumberOfCores` x `NumberOfThreadsPerCore`.

Attributes:

GetAccess	public
SetAccess	private

Data Types: uint

Examples**Specify Hardware Implementation for New Device**

This example shows how to register a new hardware device.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create("Processor", Name="MyProcessor", ...
    Manufacturer="MyManufacturer");
```

Create a `target.LanguageImplementation` object for language implementation details.

```
myLanguageImplementation = target.create("LanguageImplementation", ...
    Name="MyProcessorImplementation");
```

Specify language implementation details.

```
myLanguageImplementation.Endianness = target.Endianness.Little;
```

```
myLanguageImplementation.AtomicIntegerSize = 64;
myLanguageImplementation.AtomicFloatSize = 64;
myLanguageImplementation.WordSize = 64;
```

```
myLanguageImplementation.DataTypes.Char.Size = 8;
myLanguageImplementation.DataTypes.Short.Size = 16;
myLanguageImplementation.DataTypes.Int.Size = 32;
myLanguageImplementation.DataTypes.Long.Size = 64;
myLanguageImplementation.DataTypes.LongLong.IsSupported = true;
myLanguageImplementation.DataTypes.LongLong.Size = 64;
myLanguageImplementation.DataTypes.Float.Size = 32;
myLanguageImplementation.DataTypes.Double.Size = 64;
```

```
myLanguageImplementation.DataTypes.Pointer.Size = 32;
```

```
myLanguageImplementation.DataTypes.SizeT.Size = 64;
myLanguageImplementation.DataTypes.PtrDiffT.Size = 64;
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myLanguageImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

```
target.add summary:
```

```
    Objects added to internal database for current MATLAB session:
```

```
target.LanguageImplementation    "MyProcessorImplementation"
target.Processor               "MyManufacturer-MyProcessor"
```

- If you are using the MATLAB® Coder™: On the **Hardware** tab, you see the new device. Alternatively, you can now create a `coder.Hardware` object for this device by using the `coder.hardware` function.
- If you are using the Simulink® Coder™: On the **Hardware Implementation** pane, you can now set **Device vendor** and **Device type** to `MyManufacturer` and `MyProcessor` respectively.

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded)
```

```
target.remove summary:
```

```
Objects removed from internal database:
target.LanguageImplementation    "MyProcessorImplementation"
target.Processor               "MyManufacturer-MyProcessor"
```

Create Hardware Implementation by Modifying Existing Implementation

If an existing hardware implementation contains most of the values that you want in a new hardware implementation, you can quickly create the new implementation by creating and modifying a copy of the existing implementation.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create("Processor",Name="MyProcessor", ...
    Manufacturer="MyManufacturer");
```

Create a `target.LanguageImplementation` object that copies an existing language implementation.

```
myCopiedImplementation = target.create("LanguageImplementation", ...
    Name="MyCopiedImplementation", ...
    Copy="Atmel-AVR");
```

Specify the required language implementation details. For example, byte ordering.

```
myCopiedImplementation.Endianess = target.Endianess.Big;
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = myCopiedImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

```
target.add summary:
```

```
Objects added to internal database for current MATLAB session:
target.LanguageImplementation    "MyCopiedImplementation"
target.Processor               "MyManufacturer-MyProcessor"
```

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded)
```

```
target.remove summary:
```

```
Objects removed from internal database:
target.LanguageImplementation    "MyCopiedImplementation"
target.Processor                "MyManufacturer-MyProcessor"
```

Create Hardware Implementation by Reusing Existing Implementation

If your hardware device requires the same hardware implementation as an existing implementation, you can reuse the existing implementation.

Create a `target.Processor` object for the new hardware device.

```
myProc = target.create( "Processor", Name="MyProcessor", ...
                        Manufacturer="MyManufacturer");
```

Retrieve the existing implementation by using the identifier for the device vendor and type.

```
existingImplementation = target.get("LanguageImplementation", ...
                                   "ARM Compatible-ARM Cortex");
```

Associate the language implementation with the hardware device.

```
myProc.LanguageImplementations = existingImplementation;
```

Add the `target.Processor` object to an internal database.

```
objectsAdded = target.add(myProc);
```

```
target.add summary:
```

```
Objects added to internal database for current MATLAB session:
target.Processor                "MyManufacturer-MyProcessor"
1 object not added because they already exist.
```

To remove the objects from the internal database, enter:

```
target.remove(objectsAdded);
```

```
target.remove summary:
```

```
Objects removed from internal database:
target.Processor    "MyManufacturer-MyProcessor"
```

Create Timer Object

This example shows how you can create a timer object for your development computer.

Create the function signature for a timer. In this example, the function returns a `uint64` data type and the function name `timestamp_x86`.

```
timerSignature = target.create('Function');
timerSignature.Name = 'timestamp_x86';
timerSignature.ReturnType = 'uint64';
```

Capture the function in an API object.

```
timerApi = target.create('API');
timerApi.Functions = timerSignature;
timerApi.Language = target.Language.C;
timerApi.Name = 'Linux Timer API';
```

Capture the dependencies of the function, that is, the source and header files that are required to run the function.

```
timerDependencies = target.create('BuildDependencies');
timerDependencies.IncludeFiles = {'host_timer_x86.h'};
timerDependencies.IncludePaths = ...
    {'$(MATLAB_ROOT)/toolbox/coder/profile/src'};
timerDependencies.SourceFiles = {'host_timer_x86.c'};
```

Create an object that combines the API and dependencies.

```
timerImplementation = target.create('APIImplementation');
timerImplementation.API = timerApi;
timerImplementation.BuildDependencies = timerDependencies;
timerImplementation.Name = 'Linux Timer Implementation';
```

Create the timer object and associate it with the timer information.

```
timer = target.create('Timer');
timer.APIImplementation = timerImplementation;
timer.Name = 'Linux Timer';
```

Note Using name-value arguments, you can create the timer object with this command.

```
timer = target.create('Timer', 'Name', 'Linux Timer', ...
    'FunctionName', 'timestamp_x86', ...
    'FunctionReturnType', 'uint64', ...
    'FunctionLanguage', target.Language.C, ...
    'SourceFiles', {'host_timer_x86.c'}, ...
    'IncludeFiles', {'host_timer_x86.h'}, ...
    'IncludePaths', {'$(MATLAB_ROOT)/toolbox/coder/profile/src'})
```

Assign the timer and add-ons to the processor object.

```
processor = target.get('Processor', 'Intel-x86-64 (Linux 64)');
processor.Timers = timer;
```

Create Description for Intel Core Processor

Create a description for the Intel Core® i7-8550U processor, which is a processor that supports hyperthreading.

```
i7 = target.create('Processor', ...
    'Name', 'i7-8550U', ...
    'Manufacturer', 'Intel', ...
    'NumberOfCores', 4, ...
```



```
target.add(i7); 'NumberOfThreadsPerCore', 2);
```

Version History

Introduced in R2019a

See Also

`target.LanguageImplementation` | `target.create`

Topics

“Register New Hardware Devices”

“Remove Instrumentation Overheads by Using target Namespace”

“Generate Native Half-Precision C Code from Simulink Models” (Fixed-Point Designer)

“Specify Maximum Number of Threads to Run Parallel for-Loops in the Generated Code”

target.ProfilingFreezingOverhead Class

Namespace: target

Capture freezing and unfreezing instrumentation overhead

Description

Use a `target.ProfilingFreezingOverhead` object to capture the instrumentation overhead for freezing and unfreezing a timer.

To create a `target.ProfilingFreezingOverhead` object, use the `target.create` function.

Properties

Value — Instrumentation overhead

scalar

Specify instrumentation overhead for freezing and unfreezing a timer.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

Counter — Timer

target.Timer object

Object that provides the timer details for your processor.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

MinimumBenchmarkIterations — Instrumentation overhead

100 (default) | scalar

Specify the minimum number of iterations that benchmark program performs to compute instrumentation overhead values.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

Examples

Specify Function Instrumentation Overhead

Manually specify the function instrumentation overhead for a timer.

Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', 'myProcessorObjectId');  
timer = target.get('Timer', 'myTimerObjectId');
```

Create a `target.ProfilingFreezingOverhead` object.

```
freezingOverhead = target.create('ProfilingFreezingOverhead', ...  
                                'Name', 'Timer Freezing Overhead');  
freezingOverhead.Value = 30;  
freezingOverhead.Counter = timer;
```

Version History

Introduced in R2021a

See Also

`target.ProfilingTaskOverhead` | `target.ProfilingFunctionOverhead`

Topics

“Remove Instrumentation Overheads by Using target Namespace”

target.ProfilingFunctionOverhead Class

Namespace: target

Capture function instrumentation overhead

Description

Use a `target.ProfilingFunctionOverhead` object to capture the instrumentation overhead for profiling a function.

To create a `target.ProfilingFunctionOverhead` object, use the `target.create` function.

Properties

Value — Instrumentation overhead

scalar

Specify instrumentation overhead for profiling a function.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

Counter — Timer

target.Timer object

Object that provides the timer details for your processor.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

MinimumBenchmarkIterations — Instrumentation overhead

100 (default) | scalar

Specify the minimum number of iterations that benchmark program performs to compute instrumentation overhead values.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

Examples

Specify Function Instrumentation Overhead

Manually specify the function instrumentation overhead for a timer.

Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', 'myProcessorObjectId');  
timer = target.get('Timer', 'myTimerObjectId');
```

Create a `target.ProfilingFunctionOverhead` object.

```
functionOverhead = target.create('ProfilingFunctionOverhead', ...  
                                'Name', 'Timer Function Overhead');  
functionOverhead.Value = 20;  
functionOverhead.Counter = timer;
```

Version History

Introduced in R2021a

See Also

`target.ProfilingTaskOverhead` | `target.ProfilingFreezingOverhead`

Topics

“Remove Instrumentation Overheads by Using target Namespace”

target.ProfilingTaskOverhead Class

Namespace: target

Capture task instrumentation overhead

Description

Use a `target.ProfilingTaskOverhead` object to capture the instrumentation overhead for profiling a task.

To create a `target.ProfilingTaskOverhead` object, use the `target.create` function.

Properties

Value — Instrumentation overhead

scalar

Specify instrumentation overhead for profiling a task.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

Counter — Timer

target.Timer object

Object that provides the timer details for your processor.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

MinimumBenchmarkIterations — Instrumentation overhead

100 (default) | scalar

Specify the minimum number of iterations that benchmark program performs to compute instrumentation overhead values.

Attributes:

GetAccess	public
SetAccess	public

Data Types: int

Examples

Specify Task Instrumentation Overhead

Manually specify the task instrumentation overhead for a timer.

Retrieve the `target.Processor` and `target.Timer` objects.

```
processor = target.get('Processor', 'myProcessorObjectId');  
timer = target.get('Timer', 'myTimerObjectId');
```

Create a `target.ProfilingTaskOverhead` object.

```
taskOverhead = target.create('ProfilingTaskOverhead', ...  
                             'Name', 'Timer Task Overhead');  
taskOverhead.Value = 10;  
taskOverhead.Counter = timer;
```

Version History

Introduced in R2021a

See Also

`target.ProfilingFunctionOverhead` | `target.ProfilingFreezingOverhead`

Topics

“Remove Instrumentation Overheads by Using target Namespace”

target.remove

Namespace: target

Remove target object from internal database

Syntax

```
target.remove(targetObject)
target.remove(targetType, targetObjectId)
target.remove(targetObject, Name, Value)
```

Description

`target.remove(targetObject)` removes the target object from an internal database.

`target.remove(targetType, targetObjectId)` removes the target object specified by class and identifier.

`target.remove(targetObject, Name, Value)` uses name-value arguments to remove associated objects and suppress command-line output.

Examples

Remove Target Object From Internal Database

You can specify and add a hardware device implementation to an internal database.

```
armv8 = target.create('LanguageImplementation', ...
    'Name', 'Armv8-A LP64', ...
    'Copy', 'ARM Compatible-ARM Cortex');

a53 = target.create('Processor', ...
    'Name', 'Cortex-A53', ...
    'Manufacturer', 'ARM Compatible');

a53.LanguageImplementations = armv8;

target.add(a53)
```

When a target object is no longer required, you can use the function to remove the object from the internal database.

To remove only the `target.Processor` object, run:

```
target.remove(a53)
```

Or:

```
target.remove('Processor', 'ARM Compatible-Cortex-A53');
```


To remove the `target.Processor` object and its associated `target.LanguageImplementation` object and suppress the command-line output, run:

```
target.remove(a53, ...
              'IncludeAssociations', true, ...
              'SuppressOutput', true);
```

Input Arguments

targetObject — Target object

object

Specify the target object that you want to remove.

targetType — Target type

character vector | string

Specify the class of the target 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'`

targetObjectId — Target 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.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `target.remove(myTargetObject, 'IncludeAssociations', true);`

IncludeAssociations — Remove associated objects

false (default) | true

Remove associated objects from internal database:

- **true** -- Function removes `targetObject` and associated target objects from the internal database. If an associated object is referenced by another target object, the function does not remove the associated object. If `targetObject` is not saved in the internal database, the function does not remove associated target objects from the internal database.
- **false** -- Function removes only `targetObject` from the internal database.

Example: `target.remove(myTargetObject, 'IncludeAssociations', true);`

Data Types: logical

SuppressOutput — Control command-line output

false (default) | true

Control command-line output of function:

- **true** -- Suppress command-line output from the function.
- **false** -- Provide information about the objects that the function removes from the internal database.

Example: `target.remove(myTargetObject, 'SuppressOutput', true);`

Data Types: logical

Version History

Introduced in R2019a

See Also

`target.add` | `target.clear` | `target.create` | `target.get` | `target.update`

Topics

“Register New Hardware Devices”

target.RepeatingDirective Class

Namespace: target

Describe repeated command-line flag for tools

Description

If a command-line flag for a build tool is repeated for multiple tools, you can use a `target.RepeatingDirective` object to describe the flag. For example, multiple preprocessor definition flags can require a separate instance of the `-D` directive for each preprocessor macro being defined.

To create a `target.RepeatingDirective` object, use the `setDirective` method of the `target.BuildTool` object.

Properties

Name — Directive name

string

Name of directive.

Attributes:

GetAccess	public
SetAccess	public

Value — Directive value

string

Value of directive.

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2023a

See Also

`target.BuildTool` | `target.BuildToolType` | `target.Directive` | `target.FileType` | `target.MakefileBuilder` | `target.MakeToolType` | `target.PairedDirective` | `target.Toolchain`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.RS232Channel Class

Namespace: target

Describe serial communication channel

Description

Use the `target.RS232Channel` class, which inherits functionality from `target.CommunicationChannel`, to describe properties of the serial communication channel.

To create a `target.RS232Channel` object, use the `target.create` function.

Properties

BaudRate — Baud value

scalar integer

Baud value of the serial connection.

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2020b

See Also

`target.create` | `target.CommunicationChannel`

target.Support Class

Namespace: target

Specify association between target namespace objects

Description

Use a target.Support object to specify an association between target namespace objects. For example, you can use a target.Support object to specify that a target object runs on a target device (target.Board object) or an operating system (target.OperatingSystem object).

Class Attributes

HandleCompatible	true

For information on class attributes, see “Class Attributes”.

Creation

To create a target.Support object, use the target.create function.

Properties

Name — Object name

string scalar | character vector

Object name, specified as a string scalar or character vector.

Attributes:

GetAccess	public
SetAccess	public

For — Supported target

target object array

The supported target objects, specified as a target namespace object array.

Attributes:

GetAccess	public
SetAccess	public

RunsOn — Target object runs on

target object array

The target objects that supported target objects run on, specified as a target namespace object array.

Attributes:

GetAccess	public
SetAccess	public

Examples**Use of `target.Support` Object**

For a workflow that uses a `target.Support` object, see “Set Up PIL Connectivity for Linux-Based Target Hardware”.

Version History

Introduced in R2026a

See Also

`target.create` | `target.Processor` | `target.Board` | `target.OperatingSystem`

Topics

“Set Up PIL Connectivity for Linux-Based Target Hardware”

target.SystemCommandExecutionTool Class

Namespace: target

Capture system command information to run target application from MATLAB computer

Description

Use the `target.SystemCommandExecutionTool` to capture system command information that is required to run the target application from your development computer. Use this class to specify blocking commands. When MATLAB runs a blocking command, MATLAB waits until the command is terminated.

Properties

Name — Execution tool name

character vector | string

Name of the execution tool.

Attributes:

GetAccess	public
SetAccess	public

StartCommand — Command list to run application

target.Command object

A `target.Command` object that provides a system command for running the application. The command in the list starts the application process.

This property must not be empty.

Attributes:

GetAccess	public
SetAccess	public

StopCommand — Command list to stop application

target.Command object

A `target.Command` object that provides a system command to stop the application execution.

Attributes:

GetAccess	public
SetAccess	public

RunsOn — Context to run tool

empty `target.ExecutionContext` array (default) | `target.SSHExecutionContext` object

Context to run the execution tool, specified as an empty `target.ExecutionContext` array (default) or a `target.SSHExecutionContext` object.

Context to Run Tool	Property Value	Specified By Example
Over SSH Connection	target.SSHExecutionContext object	executionTool.RunsOn = target.ExecutionContext
Locally (default)	Empty target.ExecutionContext array	executionTool.RunsOn = target.ExecutionContext

Attributes:

GetAccess

SetAccess

public

public

Id — Object identifier

character vector | string

Value of the Name property.

Attributes:

GetAccess

SetAccess

public

private

Version History

Introduced in R2020b

See Also

target.create | target.ApplicationExecutionTool | target.Command | target.HostProcessExecutionTool

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.TargetConnection Class

Namespace: target

Provide details about connecting MATLAB computer to target hardware

Description

Use the `target.TargetConnection` class, which inherits functionality from `target.Connection`, to provide details about connecting your MATLAB computer to target hardware. For example, the communication channel and connection properties that are required for communication with your target hardware.

To create a `target.TargetConnection` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

Properties

Name — Connection object

character vector | string

Name of connection object.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

CommunicationChannel — Communication channel type

`target.CommunicationChannel` object

Associate a `target.CommunicationChannel` object with your connection, which describes the type of channel that is used. For example, to specify serial or TCP channel properties, use `target.RS232Channel` or `target.TCPChannel` respectively.

If you use name-value arguments to create a `target.TargetConnection` object, for the `CommunicationChannel` property, specify these arguments.

Name	Description
'CommunicationType'	Required. Type of predefined communication channel. Specifies one of these values: 'RS232Channel' 'TCPChannel' 'UDPChannel'
'IPAddress'	Optional. If predefined communication channel is 'TCPChannel' or 'UDPChannel', specifies <code>IPAddress</code> property of <code>target.TCPChannel</code> or <code>target.UDPChannel</code> object.

Name	Description
'Port'	Optional. If predefined communication channel is 'TCPChannel' or 'UDPChannel', specifies Port property of target.TCPChannel or target.UDPChannel object.
'BaudRate'	Optional. If predefined communication channel is 'RS232Channel', specifies BaudRate property of target.RS232Channel object.
'Parity'	Optional. If predefined communication channel is 'RS232Channel', specifies Parity property of target.RS232Channel object.

Attributes:

GetAccess	public
SetAccess	public

Target — Connected target

target.Board object

Associate a target.Board object with your connection, which describes the target hardware that is connected to the MATLAB computer.

Attributes:

GetAccess	public
SetAccess	public

ConnectionProperties — MATLAB computer connection properties

target.ConnectionProperties object

Associate a target.ConnectionProperties object with your connection that describes the MATLAB computer connection properties that are used to connect to the target hardware. For example, to specify the serial port, use a target.Port object.

If you use name-value arguments to create a target.TargetConnection object and the predefined communication channel type is 'RS232Channel', specifying the argument 'Port' sets the ConnectionProperties property to target.Port.

Attributes:

GetAccess	public
SetAccess	public

TargetConnectionProperties — Target connection properties

target.ConnectionProperties object

Associate a target.ConnectionProperties object with your connection that describes the target hardware connection properties that are used to connect to the MATLAB computer. For example, to specify the serial port, use a target.Port object.

If you use name-value arguments to create a target.TargetConnection object and the predefined communication channel type is 'RS232Channel', specifying the argument 'Port' sets the TargetConnectionProperties property to target.Port.

Attributes:

GetAccess	public
SetAccess	public

Examples**Specify Connection Between Development Computer and Target Hardware**

This code from “Set Up PIL Connectivity by Using Target Framework” shows how to specify the connection between your development computer and target hardware. In the example, the target application runs on your development computer as a separate process and uses a TCP communication channel through localhost.

```
connection = target.create('TargetConnection');
connection.Name = 'Host Process Connection';
connection.Target = hostTarget;
connection.CommunicationChannel = target.create('TCPChannel');
connection.CommunicationChannel.Name = ...
    'External Process TCPCommunicationChannel';
connection.CommunicationChannel.IPAddress = 'localhost';
connection.CommunicationChannel.Port = '0';
```

Note Using name-value arguments, you can create the connection object with this command:

```
connection = target.create('TargetConnection', ...
    'Name', 'Host Process Connection', ...
    'Target', hostTarget, ...
    'CommunicationType', 'TCPChannel', ...
    'IPAddress', 'localhost', ...
    'Port', '0')
```

Version History

Introduced in R2020b

See Also

target.create | target.CommunicationChannel | target.ConnectionProperties | target.Board

Topics

“Set Up PIL Connectivity by Using Target Framework”
 “Customize Connectivity for XCP External Mode Simulations”

target.TCPChannel Class

Namespace: target

Describe TCP communication properties

Description

Use the `target.TCPChannel`, which inherits functionality from `target.CommunicationChannel`, to describe TCP communication properties.

To create a `target.TCPChannel` object, use the `target.create` function.

Properties

IPAddress — IP address

character vector | string

IP address of the TCP server.

Attributes:

GetAccess	public
SetAccess	public

Port — Port

scalar integer

TCP port.

Attributes:

GetAccess	public
SetAccess	public

Examples

Create Connection by Using TCP Communication Channel

This code from “Set Up PIL Connectivity by Using Target Framework” shows how to specify the connection between your development computer and target hardware. In the example, the target application runs on your development computer as a separate process and uses a TCP communication channel through `localhost`.

```
connection = target.create('TargetConnection');
connection.Name = 'Host Process Connection';
connection.Target = hostTarget;
connection.CommunicationChannel = target.create('TCPChannel');
connection.CommunicationChannel.Name = ...
    'External Process TCPCommunicationChannel';
connection.CommunicationChannel.IPAddress = 'localhost';
connection.CommunicationChannel.Port = '0';
```

Note Using name-value arguments, you can create the connection object with this command:

```
connection = target.create('TargetConnection', ...  
                           'Name', 'Host Process Connection', ...  
                           'Target', hostTarget, ...  
                           'CommunicationType', 'TCPChannel', ...  
                           'IPAddress', 'localhost', ...  
                           'Port', '0')
```

Version History

Introduced in R2020b

See Also

[target.create](#) | [target.CommunicationChannel](#) | [target.TargetConnection](#)

Topics

“Set Up PIL Connectivity by Using Target Framework”

target.Timer Class

Namespace: target

Provide timer details for processor

Description

Use the `target.Timer` class to provide timer details for your processor. For example, information about the C or C++ function interface and implementation, frequency, and timer count direction. To provide information about instrumenting C or C++ code for profiling, you can associate the timer details with a `target.Processor` object.

To create a `target.Timer` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

Properties

Name — Timer name

character vector | string

Name of timer.

Attributes:

GetAccess	public
SetAccess	public

Direction — Count direction

'Up' | 'Down'

Direction of timer count.

Attributes:

GetAccess	public
SetAccess	public

APIImplementation — API implementation

target.APIImplementation object

Information about the API implementation, which is used to determine the current time.

If you use name-value arguments to create a `target.Timer` object, for the `APIImplementation` property, specify these arguments.

Name	Description
'FunctionName'	Required. Name property of <code>target.Function</code> object.
'FunctionReturnType'	Required. <code>ReturnType</code> property of <code>target.Function</code> object.

Name	Description
'IncludeFiles'	Required. IncludeFiles property of target.BuildDependencies object
'FunctionLanguage'	Optional. Language property of target.API object.
'SourceFiles'	Optional. SourceFiles property of target.BuildDependencies object.
'IncludePaths'	Optional. IncludePaths property of target.BuildDependencies object

Attributes:

GetAccess	public
SetAccess	public

Frequency — Frequency

scalar

Frequency of the unit returned by the timer function. This value can be used to convert timer function output to seconds. The helper class `target.unit.Frequency` contains some common frequency units.

Attributes:

GetAccess	public
SetAccess	public

Data Types: `uint64`

Examples

Create Timer Object

Create a timer object for your development computer.

Create the function signature for a timer. In this example, the function returns a `uint64` data type and the function name `timestamp_x86`.

```
timerSignature = target.create('Function');
timerSignature.Name = 'timestamp_x86';
timerSignature.ReturnType = 'uint64';
```

Capture the function in an API object.

```
timerApi = target.create('API');
timerApi.Functions = timerSignature;
timerApi.Language = target.Language.C;
timerApi.Name = 'Linux Timer API';
```

Capture the dependencies of the function, that is, the source and header files that are required to run the function.

```
timerDependencies = target.create('BuildDependencies');
timerDependencies.IncludeFiles = {'host_timer_x86.h'};
timerDependencies.IncludePaths = ...
    {'$(MATLAB_ROOT)/toolbox/coder/profile/src'};
timerDependencies.SourceFiles = {'host_timer_x86.c'};
```

Create an object that combines the API and dependencies.

```
timerImplementation = target.create('APIImplementation');  
timerImplementation.API = timerApi;  
timerImplementation.BuildDependencies = timerDependencies;  
timerImplementation.Name = 'Linux Timer Implementation';
```

Create the timer object and associate it with the timer information.

```
timer = target.create('Timer');  
timer.APIImplementation = timerImplementation;  
timer.Name = 'Linux Timer';
```

Note Using name-value arguments, you can create the timer object with this command.

```
timer = target.create('Timer', 'Name', 'Linux Timer', ...  
    'FunctionName', 'timestamp_x86', ...  
    'FunctionReturnType', 'uint64', ...  
    'FunctionLanguage', target.Language.C, ...  
    'SourceFiles', {'host_timer_x86.c'}, ...  
    'IncludeFiles', {'host_timer_x86.h'}, ...  
    'IncludePaths', {'$(MATLAB_ROOT)/toolbox/coder/profile/src'})
```

Assign the timer and add-ons to the processor object.

```
processor = target.get('Processor', 'Intel-x86-64 (Linux 64)');  
processor.Timers = timer;
```

Create Timer Object by Modifying Existing Object

You can create a new timer object by copying an existing timer object and modifying specific property values of the copy.

```
newTimer = target.create('Timer', ...  
    'Copy', 'Linux Timer', ...  
    'Name', 'NewTimerName', ...  
    'FunctionName', 'newFunctionName');
```

Version History

Introduced in R2020b

See Also

`target.create` | `target.APIImplementation` | `target.Processor`

target.Toolchain Class

Namespace: target

Capture high-level information about toolchain

Description

Use a `target.Toolchain` object to capture high-level information about a toolchain.

To create a `target.Toolchain` object, use the `target.create` function. Create the object in a single step.

```
toolchainObject = target.create('Toolchain', ...
                                nameOfToolchain, ...
                                parameterXName, parameterXValue, ...
                                parameterYName, parameterYValue, ...
                                parameterZName, parameterZValue)
```

nameOfToolchain is the name of the `target.Toolchain` object that you want to create.

parameterXName is the name of either a property or a referenced object property -- see tables below.

Properties

Id — Object identifier

string

Unique ID of object in internal database.

Attributes:

GetAccess	public
SetAccess	private

Name — Toolchain name

string

Name of the toolchain, in a form that you want workflows to display.

Example: 'CMake/Ninja for all hosts'

Attributes:

GetAccess	public
SetAccess	public

SupportedHardware — Supported target devices

`target.HardwareComponentSupport` object array

Optional. Target devices for which the toolchain can build code.

Attributes:

GetAccess	public
SetAccess	public

HostOperatingSystemSupport — Operating system support`target.HostOperatingSystemSupport` object

Development computer operating systems that the toolchain can run on.

Attributes:

GetAccess	public
SetAccess	public

Tools — Toolchain tools`target.Tool` object array

Tools in the toolchain.

If you use name-value arguments to create a `target.Toolchain` object, for the `Tools` property, specify these arguments.

Name	Description
'Family'	Provides an initial list of tools through a default set of <code>target.BuildTool</code> objects configured for the specified family. Example: 'Family', 'GNU'
'Assembler'	Optional. In the list of tools, specify the assembler through a <code>target.BuildTool</code> object. The argument sets: - Name of <code>BuildToolType</code> property (a <code>target.BuildToolType</code> object) to 'Assembler'. - String and Arguments of <code>Command</code> property (a <code>target.Command</code> object) using specified value. Example: 'Assembler', 'as'
'CCompiler'	Optional. In the list of tools, specify the C compiler through a <code>target.BuildTool</code> object. The argument sets: - Name of <code>BuildToolType</code> property (a <code>target.BuildToolType</code> object) to 'C Compiler'. - String and Arguments of <code>Command</code> property (a <code>target.Command</code> object) using specified value. Example: 'CCompiler', 'gcc'

Name	Description
'CppCompiler'	Optional. In the list of tools, specify the C++ compiler through a <code>target.BuildTool</code> object. The argument sets: • Name of <code>BuildToolType</code> property (a <code>target.BuildToolType</code> object) to 'C++ Compiler'. • String and Arguments of Command property (a <code>target.Command</code> object) using specified value. Example: 'CppCompiler', 'g++'
'Linker'	Optional. In the list of tools, specify the linker through a <code>target.BuildTool</code> object. The argument sets: • Name of <code>BuildToolType</code> property (a <code>target.BuildToolType</code> object) to 'Linker'. • String and Arguments of Command property (a <code>target.Command</code> object) using specified value. Example: 'Linker', 'gcc'
'CppLinker'	Optional. In the list of tools, specify the C++ linker through a <code>target.BuildTool</code> object. The argument sets: • Name of <code>BuildToolType</code> property (a <code>target.BuildToolType</code> object) to 'C++ Linker'. • String and Arguments of Command property (a <code>target.Command</code> object) using specified value. Example: 'CppLinker', 'g++'
'Archiver'	Optional. In the list of tools, specify the archiver through a <code>target.BuildTool</code> object. The argument sets: • Name of <code>BuildToolType</code> property (a <code>target.BuildToolType</code> object) to 'Archiver'. • String and Arguments of Command property (a <code>target.Command</code> object) using specified value. Example: 'Archiver', 'ar ruvs'
'MakeTool'	Optional. In the list of tools, specify the make tool through a <code>target.BuildTool</code> object. The argument sets: • Name of <code>BuildToolType</code> property (a <code>target.BuildToolType</code> object) to 'Make Tool'. • String and Arguments of Command property (a <code>target.Command</code> object) using specified value. Example: 'MakeTool', 'mingw32-gmake -j\$(NUM_CORES)'

Name	Description
'CommandFile'	Optional. For each tool that supports a command file directive, specify the directive through the corresponding <code>target.BuildTool</code> object. The argument sets: • Name of Directive property (a <code>target.Directive</code> object) to 'CommandFile'. • String of Directive property to specified value. Example: 'CommandFile', '@'
'ObjectExtension'	Optional. Specify the file extension that the tools (<code>target.BuildTool</code> objects) use for object files. Example: 'ObjectExtension', '.obj'
'ExecutableExtension'	Optional. Specify the file extension that the tools (<code>target.BuildTool</code> objects) use for executable files. Example: 'ExecutableExtension', '.exe'
'SharedLibraryExtension'	Optional. Specify the file extension that the tools (<code>target.BuildTool</code> objects) use for shared library files. Example: 'SharedLibraryExtension', '.dll'
'StaticLibraryExtension'	Optional. Specify the file extension that the tools (<code>target.BuildTool</code> objects) use for static library files. Example: 'StaticLibraryExtension', '.a'
'FileSep'	Optional. Specify the file separator that the tools (<code>target.BuildTool</code> objects) use. Example: 'FileSep', '\'
'ToolPrefix'	Optional when 'Family' name-value argument is used. Specify prefix for default command names in <code>target.BuildTool</code> objects created for the family. Example: 'ToolPrefix', 'arm-none-eabi-'

Attributes:

GetAccess	public
SetAccess	public

Builder — Builder configuration
`target.CMakeBuilder` object | `target.MakefileBuilder` object

An object that describes the configuration of a build automation tool. For a CMake-based toolchain, use a `target.CMakeBuilder` object. For a makefile-based toolchain, use a `target.MakefileBuilder` object.

If you use name-value arguments to create a `target.Toolchain` object, for the `Builder` property, specify this argument.

Name	Description
'MakeToolType'	Required. MakeToolType property of target.MakefileBuilder object. Value must be 'GMake', 'NMake', or 'CMake'. Example: 'MakeToolType', 'GMake'
'Generator'	Optional. Generator property of target.CMakeBuilder object. Example: 'Generator', 'Unix Makefiles'
'ToolchainFile'	Optional. ToolchainFile property of target.CMakeBuilder object. Example: 'ToolchainFile', 'pathToFolder/cmake/myPlatform.cmake'

Attributes:

GetAccess	public
SetAccess	public

BuildRequirements — Build dependencies

target.BuildDependencies object

C and C++ build dependencies that you must associate with the toolchain. For example, specific preprocessor directives or libraries. When you specify this property, every build that uses the toolchain includes the dependencies.

If you use name-value arguments to create a target.Toolchain object, for the BuildRequirements property, specify these arguments.

Name	Description
'SharedLibraries'	Optional. SharedLibraries property of target.BuildDependencies object. Example: 'SharedLibraries', {'m', 'ws2_32'}
'CompilerFlags'	Optional. CompilerFlags property of target.BuildDependencies object. Example: 'CompilerFlags', {'-fwrapv', '-fPIC'}
'LinkerFlags'	Optional. LinkerFlags property of target.BuildDependencies object. Example: 'LinkerFlags', {'/DEBUG'}

Attributes:

GetAccess	public
SetAccess	public

EnvironmentConfiguration — Environment configuration

target.EnvironmentConfiguration object array

An array of objects that provide system environment setup commands and paths that are required for the use of the toolchain. For example, vendor-supplied setup scripts. Use only one object for each operating system of a development computer.

If you use name-value arguments to create a `target.Toolchain` object, for the `EnvironmentConfiguration` property, specify this argument.

Name	Description
'SystemPaths'	Optional. <code>SystemPaths</code> property of <code>target.EnvironmentConfiguration</code> object. Example: 'SystemPaths', {'\$(MW_MINGW64_LOC)/bin'}

Attributes:

GetAccess	public
SetAccess	public

RunsOn — Build context

empty `target.ExecutionContext` array (default) | `target.SSHExecutionContext` object | `target.WSLExecutionContext`

Build context, specified as an empty `target.ExecutionContext` array (default), a `target.SSHExecutionContext` object, or a `target.WSLExecutionContext` object.

Build Context	Property Value	Example
Remote, using SSH connection	<code>target.SSHExecutionContext</code> object	<code>toolChainObj.RunsOn = target.ExecutionContext</code>
Windows Subsystem for Linux (WSL) environment	<code>target.WSLExecutionContext</code> object	<code>toolChainObj.RunsOn = target.ExecutionContext</code>
Local (default)	Empty <code>target.ExecutionContext</code> array	<code>toolChainObj.RunsOn = target.ExecutionContext</code>

Attributes:

GetAccess	public
SetAccess	public

Methods

Public Methods

`getBuildToolOfType` Search for type of build tool

Examples

Use target.Toolchain to Configure CMake Build

This code snippet from “Create Custom CMake Toolchain Definition” shows how you can create a `target.Toolchain` object and use it to configure the way CMake builds generated code.

```
tc = target.create('Toolchain', 'Name', 'Example Custom CMake Toolchain');
tc.Builder = target.create('CMakeBuilder');
tc.Builder.Generator = 'Ninja';
tc.Builder.ToolchainFile = 'pathTo/ExampleCMakeToolchain.cmake';
```

Or, create the object in a single step.

```
tc = target.create('Toolchain', ...
    'Name', 'Example Custom CMake Toolchain', ...
    'MakeToolType', 'CMake', ...
    'Generator', 'Ninja', ...
    'ToolchainFile', 'pathTo/ExampleCMakeToolchain.cmake')
```

Use target.Toolchain Object to Configure Makefile Build

This command creates an object for a GNU-based toolchain by using several name-value arguments.

```
armtc = target.create('Toolchain', ...
    'MakeToolType', 'GMake', ...
    'Name', 'ARM Toolchain', ...
    'SystemPaths', {'$(ARM_TOOLS_LOCATION)/bin'}, ...
    'FileSep', '/', ...
    'CCompiler', 'arm-none-eabi-gcc', ...
    'CppCompiler', 'arm-none-eabi-g++', ...
    'Linker', 'arm-none-eabi-gcc', ...
    'CppLinker', 'arm-none-eabi-g++', ...
    'CompilerFlags', {'-fwrapv', '-fPIC'});
```

Using the Family and ToolPrefix name-value arguments, you can create the same object.

```
target.create('Toolchain', ...
    'Family', 'GNU', ...
    'Name', 'ARM Toolchain', ...
    'ToolPrefix', 'arm-none-eabi-');
```

To capture information about a more customized toolchain, you can use additional name-value arguments. For example:

```
target.create('Toolchain', ...
    'Family', 'GNU', ...
    'Name', 'ARM Toolchain', ...
    'ToolPrefix', 'arm-none-eabi-', ...
    'SystemPaths', {'$(ARM_TOOLS_LOCATION)/bin'}, ...
    'CompilerFlags', {'-fwrapv', '-fPIC'});
```

Version History

Introduced in R2022b

See Also

target.BuildTool | target.BuildToolType | target.CMake | target.CMakeBuildType |
 target.CMakeBuilder | target.CMakeCacheEntry | target.Directive |
 target.EnvironmentConfiguration | target.FileType |
 target.HardwareComponentSupport | target.MakefileBuilder | target.MakeToolType |
 target.PairedDirective | target.RepeatingDirective | target.create

Topics

“Create Custom CMake Toolchain Definition”

“Build Code in Windows Subsystem for Linux”

“Define Custom Makefile-Based Toolchains Using Target Framework”

getBuildToolOfType

Search for type of build tool

Syntax

```
buildToolObj=myToolchain.getBuildToolOfType(toolType)
```

Description

`buildToolObj=myToolchain.getBuildToolOfType(toolType)` searches for the specified tool type in the `Tools` property of the `target.Toolchain` object and returns a `target.BuildTool` object.

Input Arguments

toolType — Tool type search

character vector | string scalar

Specify tool type to search for. Provided the tool is defined in the toolchain, you can specify one of these options:

- 'Assembler'
- 'C Compiler'
- 'C++ Compiler'
- 'Archiver'
- 'Linker'
- 'C++ Linker'
- 'Make Tool'
- 'HEX Converter'
- 'BIN Converter'
- 'Executable Size'

Output Arguments

buildToolObj — Build tool object

`target.BuildTool` object

`target.BuildTool` object that describes the build tool.

Examples

Specify Extensions for Build Tool Files

This code snippet shows how you can specify file extensions for a tool (for example, the assembler) in a toolchain.

```
tc = target.create('Toolchain', ...  
                  'Name', 'myToolchain', ...  
                  'Family', 'GNU', ...  
                  'ToolPrefix', 'powerpc64-linux-gnu');  
asm = tc.getBuildToolOfType('Assembler');  
asm.setFileExtensions('Source', {'.s', '.S', '.asm'});
```

Version History

Introduced in R2026a

See Also

`target.Toolchain` | `target.BuildTool` | `target.BuildToolType`

Topics

“Define Custom Makefile-Based Toolchains Using Target Framework”

target.Tools Class

Namespace: target

Describe properties of tools for target hardware

Description

Use the `target.Tools` class to capture properties of tools that enable your development computer to interact with the target hardware.

Properties

DebugTools — Supported debuggers

`target.ApplicationExecutionTool` object vector

Descriptions of debugging tools that are supported for the target hardware.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

ExecutionTools — Supported execution tools

`target.ApplicationExecutionTool` object vector

Descriptions of supported tools for executing applications on the target hardware.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Examples

PIL Target Connectivity with Debugger

For an example that uses the `target.Tools` class, see “Use Debugger for PIL Target Connectivity”.

Version History

Introduced in R2020b

See Also

`target.create` | `target.Board`

Topics

“Use Debugger for PIL Target Connectivity”

target.UDPChannel Class

Namespace: target

Describe UDP communication

Description

Use the `target.UDPChannel` class, which inherits functionality from `target.CommunicationChannel`, to describe User Datagram Protocol (UDP) communication.

To create a `target.UDPChannel` object, use the `target.create` function.

Properties

IPAddress — IP address

character vector | string

IP address of the UDP server.

Attributes:

GetAccess	public
SetAccess	public

Port — Port

scalar integer

UDP port.

Attributes:

GetAccess	public
SetAccess	public

Version History

Introduced in R2020b

See Also

`target.create` | `target.CommunicationChannel` | `target.TargetConnection`

target.update

Namespace: target

Update target objects in internal database

Syntax

```
target.update(targetObject)
```

Description

`target.update(targetObject)` finds an object in the internal database that matches `targetObject`, and updates its properties. If the object in the database is associated with child objects, the child object definitions must match the corresponding child object definitions for `targetObject`. Otherwise, the function does not perform the update and produces an error.

Examples

Update Modified Target Object in Internal Database

This example shows how you can modify a target object in the MATLAB workspace and then update the corresponding object definition in the internal database.

Create a new target object.

```
myProcObject = target.create('Processor', 'Name', 'MyProcessor')
```

```
myProcObject =
```

```
    Processor with identifier "MyProcessor" and properties:
```

```
    LanguageImplementations: [0x0 target.LanguageImplementation]
        NumberOfCores: 1
        NumberOfLogicalCores: 1
        NumberOfThreadsPerCore: 1
            Overheads: [0x0 target.Overhead]
            Timers: [0x0 target.Counter]
        Manufacturer: ''
            AddOns: [0x0 target.AddOn]
                Id: 'MyProcessor'
                Name: 'MyProcessor'
            IsValid: 1
```

Add the object to the internal database.

```
addedObject = target.add(myProcObject);
```

```
target.add summary:
```

```
    Objects added to internal database for current MATLAB session:
        target.Processor    "MyProcessor"
```

In the MATLAB workspace, modify a property of the object.

```
myProcObject.NumberOfCores = 2;
```

Update the object definition in the internal database.

```
target.update(myProcObject);
```

```
Target objects updated.
```

To verify that the object definition is updated in the internal database, retrieve the object from the internal database.

```
retrievedObject = target.get('Processor', myProcObject.Id)
```

```
retrievedObject =
```

```
    Processor with identifier "MyProcessor" and properties:
```

```
    LanguageImplementations: [0x0 target.LanguageImplementation]
           NumberOfCores: 2
    NumberOfLogicalCores: 2
    NumberOfThreadsPerCore: 1
           Overheads: [0x0 target.Overhead]
           Timers: [0x0 target.Counter]
    Manufacturer: ''
           AddOns: [0x0 target.AddOn]
              Id: 'MyProcessor'
              Name: 'MyProcessor'
           IsValid: 1
```

Input Arguments

targetObject — Updated properties object

object scalar or vector

Workspace objects that contain updated properties for objects in the internal database.

Example: `target.update(myTargetObject)`

Version History

Introduced in R2023a

See Also

`target.add` | `target.create`

Topics

“Register New Hardware Devices”

target.upgrade

Namespace: target

Upgrade existing definitions of hardware devices

Syntax

```
target.upgrade(upgraderForRegistrationMechanism, pathToRegistrationFile)
target.upgrade(___, Name, Value)
```

Description

`target.upgrade(upgraderForRegistrationMechanism, pathToRegistrationFile)` uses an upgrade procedure to create objects from data definitions in current artifacts. The function creates `registerUpgradedTargets.m` in the current working folder.

To register the upgraded data definitions with MATLAB, run `registerUpgradedTargets()`.

For persistence across MATLAB sessions, run `registerUpgradedTargets('UserInstall', true)`.

`target.upgrade(___, Name, Value)` specifies additional options using one or more name-value pair arguments.

Examples

Upgrade Data Definitions for Hardware Devices

To upgrade existing hardware device definitions that are specified through `rtwTargetInfo.m` file or `sl_customization.m` file (only applies to Simulink® Coder™ and Embedded Coder®), use the `target.upgrade` function.

rtwTargetInfo.m File

Suppose you have the hardware device definition in an `rtwTargetInfo.m` file:

```
function rtwTargetInfo(tr)
    % Add registration function handle to the Target Registry
    tr.registerTargetInfo(@loc_register_hardware);
end

function hw = loc_register_hardware
    hw = RTW.HWDeviceRegistry;
    hw.Vendor = 'MyManufacturer';
    hw.Type = 'MyDevice';
    hw.Alias = {};
    hw.Platform = {'Prod', 'Target'};
    hw.setWordSizes([8 16 32 64 64 64 64 64 64 64]);
    hw.Endianess = 'Little';
    hw.IntDivRoundTo = 'Zero';
```

```
hw.ShiftRightIntArith = true;
hw.LargestAtomicInteger = 'Long';
hw.LargestAtomicFloat = 'Double';
end
```

To upgrade the data definitions contained in the file, enter:

```
target.upgrade("rtwTargetInfo", "myPathTo/rtwTargetInfo.m");
```

In the current folder, the function creates this `registerUpgradedTargets.m` file:

```
function processor = registerUpgradedTargets(varargin)
% This function was generated using target data export.

% Create target.LanguageImplementation 'MyManufacturer-MyDevice'
languageimplementation = target.create('LanguageImplementation');
languageimplementation.AtomicFloatSize = 64;
languageimplementation.AtomicIntegerSize = 64;
languageimplementation.DataTypes.Char.Size = 8;
languageimplementation.DataTypes.Double.Size = 64;
languageimplementation.DataTypes.Float.Size = 64;
languageimplementation.DataTypes.Half.IsSupported = false;
languageimplementation.DataTypes.Half.Size = 16;
languageimplementation.DataTypes.Int.Size = 32;
languageimplementation.DataTypes.Long.Size = 64;
languageimplementation.DataTypes.LongLong.IsSupported = false;
languageimplementation.DataTypes.LongLong.Size = 64;
languageimplementation.DataTypes.Pointer.Size = 64;
languageimplementation.DataTypes.PtrDiffT.Size = 64;
languageimplementation.DataTypes.Short.Size = 16;
languageimplementation.DataTypes.SizeT.Size = 64;
languageimplementation.Name = 'MyManufacturer-MyDevice';
languageimplementation.WordSize = 64;

% Create target.Processor 'MyManufacturer-MyDevice'
processor = target.create('Processor');
processor.LanguageImplementations(1) = languageimplementation;
processor.Manufacturer = 'MyManufacturer';
processor.Name = 'MyDevice';

% Add the target objects to MATLAB memory
target.add(processor, varargin{:});
end
```

To register the hardware device with MATLAB®, enter:

```
registerUpgradedTargets()
```

If you want the registration to persist across MATLAB sessions, enter:

```
registerUpgradedTargets(UserInstall=true)
```

sl_customization.m File (only applies to Simulink Coder and Embedded Coder)

Suppose you have multiple hardware device definitions in an `sl_customization.m` file:

```
function sl_customization(cm)
% sl_customization function to register a device
% vendor and type with Simulink.
```



```

% Copy or rename this file to sl_customization.m.
cm.registerTargetInfo(@loc_register_device);
cm.registerTargetInfo(@loc_register_device2);
cm.registerTargetInfo(@loc_createConfig);

cm.registerTargetInfo(@locRegisterTfl);
cm.CodeCoverageTools.add('DummyCoverageToolForTesting',...
                        'HDummyCovTool',...
                        'A Coverage Tool Vendor');

end

function thisDev = loc_register_device
    thisDev = RTW.HWDeviceRegistry;
    thisDev.Vendor = 'MyDevVendor';
    thisDev.Type = 'MyDevType';
    thisDev.Alias = {};
    thisDev.Platform = {'Prod', 'Target'};
    thisDev.setWordSizes([8 16 32 32 32]);
    thisDev.LargestAtomicInteger = 'Char';
    thisDev.LargestAtomicFloat = 'None';
    thisDev.Endianess = 'Unspecified';
    thisDev.IntDivRoundTo = 'Undefined';
    thisDev.ShiftRightIntArith = true;
    thisDev.setEnabled({'IntDivRoundTo'});
end

function thisDev = loc_register_device2
    thisDev = RTW.HWDeviceRegistry;
    thisDev.Vendor = 'MyDevVendor';
    thisDev.Type = 'MyDevType2';
    thisDev.Alias = {};
    thisDev.Platform = {'Prod', 'Target'};
    thisDev.setWordSizes([8 16 32 32 32]);
    thisDev.LargestAtomicInteger = 'Char';
    thisDev.LargestAtomicFloat = 'None';
    thisDev.Endianess = 'Unspecified';
    thisDev.IntDivRoundTo = 'Undefined';
    thisDev.ShiftRightIntArith = true;
    thisDev.setEnabled({'IntDivRoundTo'});
end

% local function
function config = loc_createConfig
    config = rtw.connectivity.ConfigRegistry;
    config.ConfigName = 'Infineon->C16x, XC16x';
    config.ConfigClass = 'pil_slcust.HostDemoConfig1';
    config.SystemTargetFile = {'custom_target.tlc'};
    config.TemplateMakefile = {'custom_target.tmf'};
    config.TargetHWDeviceType = {'Infineon->C16x, XC16x'};
end

function thisTfl = locRegisterTfl
    thisTfl(1) = RTW.TflRegistry;
    thisTfl(1).Name = 'myTFL1';
    thisTfl(1).Description = 'Test';
    thisTfl(1).TableList = {'tfl_table_Sum',...
                           'tfl_table_Product',...
                           }; % Sum includes Add and Subtract

```

```
thisTfl(1).BaseTfl = 'ANSI_C';  
thisTfl(1).TargetHWDeviceType = {'*'};  
end
```

To upgrade the RTW.HWDeviceRegistry data definitions in the file, enter:

```
target.upgrade("sl_customization","myPathTo/sl_customization.m")
```

In the current folder, the function creates this registerUpgradedTargets.m file:

```
function targetObjects = registerUpgradedTargets(varargin)  
% This function was generated using target data export.  
  
% Create target.LanguageImplementation 'MyDevVendor-MyDevType'  
languageimplementation = target.create('LanguageImplementation');  
languageimplementation.AtomicIntegerSize = 8;  
languageimplementation.DataTypes.Char.Size = 8;  
languageimplementation.DataTypes.Double.Size = 64;  
languageimplementation.DataTypes.Float.Size = 32;  
languageimplementation.DataTypes.Half.IsSupported = false;  
languageimplementation.DataTypes.Half.Size = 16;  
languageimplementation.DataTypes.Int.Size = 32;  
languageimplementation.DataTypes.Long.Size = 32;  
languageimplementation.DataTypes.LongLong.IsSupported = false;  
languageimplementation.DataTypes.LongLong.Size = 64;  
languageimplementation.DataTypes.Pointer.Size = 32;  
languageimplementation.DataTypes.PtrDiffT.Size = 32;  
languageimplementation.DataTypes.Short.Size = 16;  
languageimplementation.DataTypes.SizeT.Size = 32;  
languageimplementation.Endianess = target.Endianess.Unspecified;  
languageimplementation.Name = 'MyDevVendor-MyDevType';  
languageimplementation.WordSize = 32;  
  
% Create target.Processor 'MyDevVendor-MyDevType'  
processor = target.create('Processor');  
processor.LanguageImplementations(1) = languageimplementation;  
processor.Manufacturer = 'MyDevVendor';  
processor.Name = 'MyDevType';  
  
% Create target.LanguageImplementation 'MyDevVendor-MyDevType2'  
languageimplementation2 = target.create('LanguageImplementation');  
languageimplementation2.AtomicIntegerSize = 8;  
languageimplementation2.DataTypes.Char.Size = 8;  
languageimplementation2.DataTypes.Double.Size = 64;  
languageimplementation2.DataTypes.Float.Size = 32;  
languageimplementation2.DataTypes.Half.IsSupported = false;  
languageimplementation2.DataTypes.Half.Size = 16;  
languageimplementation2.DataTypes.Int.Size = 32;  
languageimplementation2.DataTypes.Long.Size = 32;  
languageimplementation2.DataTypes.LongLong.IsSupported = false;  
languageimplementation2.DataTypes.LongLong.Size = 64;  
languageimplementation2.DataTypes.Pointer.Size = 32;  
languageimplementation2.DataTypes.PtrDiffT.Size = 32;  
languageimplementation2.DataTypes.Short.Size = 16;  
languageimplementation2.DataTypes.SizeT.Size = 32;  
languageimplementation2.Endianess = target.Endianess.Unspecified;  
languageimplementation2.Name = 'MyDevVendor-MyDevType2';  
languageimplementation2.WordSize = 32;
```

```

% Create target.Processor 'MyDevVendor-MyDevType2'
processor2 = target.create('Processor');
processor2.LanguageImplementations(1) = languageimplementation2;
processor2.Manufacturer = 'MyDevVendor';
processor2.Name = 'MyDevType2';

targetObjects = [processor, processor2];

% Add the target objects to MATLAB memory
target.add(targetObjects, varargin{:});
end

```

To register the hardware device definitions with MATLAB, enter:

```
registerUpgradedTargets()
```

If you want the registration to persist across MATLAB sessions, enter:

```
registerUpgradedTargets(UserInstall=true)
```

Input Arguments

upgraderForRegistrationMechanism — Upgrade procedure

'rtwTargetInfo' | 'sl_customization'

Select upgrade procedure for current registration mechanism.

pathToRegistrationFile — Full file name

character vector | string

Specify file that contains current registration mechanism.

Example: `target.upgrade('rtwTargetInfo', 'myPath/mySubfolder/rtwTargetInfo.m')`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `target.upgrade('rtwTargetInfo', 'myPath/mySubfolder/rtwTargetInfo.m', 'ExportFileName', 'myNewFile')`

ExportToMATLABFunction — Export to MATLAB

true (default) | false

- `true` -- Generate a MATLAB function that registers the upgraded definitions using `target.add`.
- `false` -- Do not generate a function.

ExportFileName — Generated function file name

'registerUpgradedTargets.m' (default) | string

If `ExportToMATLABFunction` is `true`, the argument specifies the file name of the generated MATLAB function. Otherwise, the argument is ignored.

Overwrite — Overwrite existing file

false (default) | true

- `true` -- If the file specified by `ExportFileName` exists, overwrite the file.
- `false` -- If the file specified by `ExportFileName` exists, the function produces an error.

If `ExportToMATLABFunction` is `false`, the argument is ignored.

Version History

Introduced in R2019b

See Also

`target.add` | `target.create`

Topics

“Register New Hardware Devices”

target.XCP Class

Namespace: target

Describe XCP protocol stack for target hardware

Description

Use the `target.XCP` class to describe the XCP protocol stack for the target hardware.

To create a `target.XCP` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

Properties

XCPTransport — XCP transport protocol

`target.XCPTransport` object

Specify the XCP transport protocol layer through a `target.XCPTCIPTransport` or a `target.XCPSerialTransport` object.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

XCPPlatformAbstraction — XCP platform abstraction

`target.XCPPlatformAbstraction` object

Optional property to specify the XCP platform abstraction layer. If you do not specify a value, the software uses the default platform abstraction layer, which supports Linux, Windows, and Mac platforms.

Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

Examples

XCP Protocol Stack for Target Hardware

This code snippet from “Customize Connectivity for XCP External Mode Simulations” shows how you can provide a description of the XCP protocol stack for your target hardware.

```
xcpPlatformAbstraction = target.create('XCPPlatformAbstraction', ...
    'Name', 'XCP Platform Abstraction');

customPlatformAbstractionPath = 'pathToImplementationFolder';
xcpPlatformAbstraction.BuildDependencies.SourceFiles = ...
    {fullfile(customPlatformAbstractionPath, 'myXCPPlatform.c')};
xcpPlatformAbstraction.BuildDependencies.IncludePaths = ...
```

```
        {customPlatformAbstractionPath});

xcpTransport = target.create('XCPTCPIPTransport', ...
    'Name', 'XCP Transport');
xcpConfiguration = target.create('XCP', ...
    'Name', 'XCP Configuration', ...
    'XCPTTransport', xcpTransport, ...
    'XCPPlatformAbstraction', xcpPlatformAbstraction);
```

Note You can create the `target.XCP` object in a single step.

```
xcpConfiguration = target.create('XCP', ...
    'Name', 'XCP TCP/IP Configuration', ...
    'XCPTTransportLayer', 'TCP', ...
    'SourceFiles', {fullfile('pathToImplementationFolder', 'myXCPPlatform.c')}, ...
    'IncludePaths', {'pathToImplementationFolder'});
```

Version History

Introduced in R2021a

See Also

`target.create` | `target.XCPTransport` | `target.XCPPlatformAbstraction`

Topics

“Customize Connectivity for XCP External Mode Simulations”

target.XCPExternalModeConnectivity Class

Namespace: target

Represent connectivity options in external mode protocol stack

Description

Use the `target.XCPExternalModeConnectivity` class, which is derived from `target.ExternalModeConnectivity`, to represent XCP connectivity options in the external mode protocol stack.

To create a `target.XCPExternalModeConnectivity` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

Properties

XCP — XCP configuration

target.XCP object

Specify XCP protocol stack for target hardware.

Examples

Specify External Mode Protocol Stack for Target Hardware

This code snippet from “Customize Connectivity for XCP External Mode Simulations” shows how to specify the external mode protocol stack for your target hardware.

```
xcpPlatformAbstraction = target.create('XCPPlatformAbstraction', ...
    'Name', 'XCP Platform Abstraction');

xcpPlatformAbstraction.BuildDependencies.Defines = {'XCP_CUSTOM_PLATFORM'};
customPlatformAbstractionPath = 'pathToImplementationFolder';
xcpPlatformAbstraction.BuildDependencies.SourceFiles = ...
    {fullfile(customPlatformAbstractionPath, 'myXCPPlatform.c')};
xcpPlatformAbstraction.BuildDependencies.IncludePaths = ...
    {customPlatformAbstractionPath};

xcpTransport = target.create('XCPTCPIPTransport', ...
    'Name', 'XCP Transport');

xcpConfiguration = target.create('XCP', ...
    'Name', 'XCP Configuration', ...
    'XCPTCPIPTransport', xcpTransport, ...
    'XCPPlatformAbstraction', xcpPlatformAbstraction);

extModeTCPConnectivity = ...
    target.create('XCPExternalModeConnectivity', ...
        'Name', 'External Mode TCP Connectivity', ...
        'XCP', xcpConfiguration);

externalMode = target.create('ExternalMode', ...
    'Name', 'External Mode', ...
    'Connectivities', extModeTCPConnectivity);
```

```
board.CommunicationProtocolStacks = externalMode;
```

Note You can create the `target.XCPExternalModeConnectivity` object in a single step.

```
extModeTCPConnectivity = target.create('XCPExternalModeConnectivity', ...  
    'Name', 'External Mode TCP Connectivity', ...  
    'XCPTransportLayer', 'TCP', ...  
    'Defines', {'XCP_CUSTOM_PLATFORM'}, ...  
    'SourceFiles', {fullfile('pathToImplementationFolder', 'myXCPPlatform.c')}, ...  
    'IncludePaths', {'pathToImplementationFolder'});
```

Version History

Introduced in R2021a

See Also

`target.create` | `target.ExternalMode`

Topics

“Customize Connectivity for XCP External Mode Simulations”

target.XCPPlatformAbstraction Class

Namespace: target

Specify XCP platform abstraction layer for target hardware

Description

Use the `target.XCPPlatformAbstraction` class to specify the implementation of the “XCP Platform Abstraction Layer” for your target hardware. The layer provides:

- The implementation of a static memory allocator -- see “Memory Allocator”.
- Other target hardware-specific functionality -- see “Other Platform Abstraction Layer Functionality”.

To create a `target.XCPPlatformAbstraction` object, use the `target.create` function. Create the object and then use separate steps to specify properties. Or, using name-value arguments, create the object and specify properties in a single step.

If you do not create a `target.XCPPlatformAbstraction` object, the software uses the default platform abstraction layer, which supports Linux, Windows, and Mac platforms.

Properties

BuildDependencies — Platform abstraction layer build dependencies

`target.BuildDependencies` object

Specify preprocessor directives, source files, and header files that are required for the implementation of the XCP platform abstraction layer.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Examples

Specify Communication Protocol Stack for Target Hardware

This code snippet from “Customize Connectivity for XCP External Mode Simulations” shows how to specify and use a custom implementation of the XCP platform abstraction layer.

```
xcpPlatformAbstraction = target.create('XCPPlatformAbstraction', ...
    'Name', 'XCP Platform Abstraction');

customPlatformAbstractionPath = 'pathToImplementationFolder';
xcpPlatformAbstraction.BuildDependencies.SourceFiles = ...
    {fullfile(customPlatformAbstractionPath, 'myXCPPlatform.c')};
xcpPlatformAbstraction.BuildDependencies.IncludePaths = ...
    {customPlatformAbstractionPath};

xcpTCPIPTransport = target.create('XCPTCIPTransport', ...
    'Name', 'XCP TCPIP Transport');
```

```
xcpTCPIPConfiguration = target.create('XCP', ...  
    'Name', 'XCP TCP/IP Configuration', ...  
    'XCPTransport', xcpTCPIPTransport, ...  
    'XCPPlatformAbstraction', xcpPlatformAbstraction);
```

Note You can create the `target.XCPPlatformAbstraction` object in a single step.

```
xcpPlatformAbstraction = target.create('XCPPlatformAbstraction', ...  
    'Name', 'XCP Platform Abstraction', ...  
    'SourceFiles', {fullfile('pathToImplementationFolder', 'myXCPPlatform.c')}, ...  
    'IncludePaths', {'pathToImplementationFolder'});
```

Version History

Introduced in R2021a

See Also

`target.create` | `target.XCP`

Topics

“Customize Connectivity for XCP External Mode Simulations”

target.XCPTCPIPTransport Class

Namespace: target

Represent XCP TCP/IP transport protocol layer

Description

Use the `target.XCPTCPIPTransport` class, which inherits functionality from `target.XCPTTransport`, to represent the XCP TCP/IP transport protocol layer for your target hardware.

Examples

XCP Protocol Stack for Target Hardware

This code snippet shows how you can use the `target.XCPTCPIPTransport` class to represent the XCP TCP/IP transport protocol layer for your target hardware.

```
xcpTCPIPTransport = ...  
    target.create('XCPTCPIPTransport', ...  
        'Name', 'XCP TCP/IP Transport');
```

Version History

Introduced in R2021a

See Also

`target.create` | `target.XCP` | `target.XCPTTransport`

Topics

“Customize Connectivity for XCP External Mode Simulations”

target.XCPTransport Class

Namespace: target

Base class for XCP transport protocol layer

Description

The `target.XCPTransport` class is a base class for representing the XCP transport protocol layer on the target hardware. The `target.XCPTCPIPTransport` and `target.XCPSerialTransport` classes are derived from this class.

Class Attributes

Abstract	true
HandleCompatible	true

For information on class attributes, see “Class Attributes”.

Properties

MaxCTOSize — Maximum CTO size

255 (default) | scalar integer

Maximum size of CTO packet, in bytes.

Example: `extModeConnectivity.XCP.XCPTransport.MaxCTOSize = 16`

Attributes:

GetAccess	public
SetAccess	public

Data Types: `uint8`

MaxDTOSize — Maximum DTO size

65532 (default) | scalar integer

Maximum size of DTO packet, in bytes.

Example: `extModeConnectivity.XCP.XCPTransport.MaxDTOSize = 256`

Attributes:

GetAccess	public
SetAccess	public

Data Types: `uint16`

MaxODTEntrySizeDAQ — Maximum ODT size

255 (default) | scalar integer

Maximum size of an ODT entry in a DAQ list.

Example: `extModeConnectivity.XCP.XCPTransport.MaxODTEntrySizeDAQ = 128`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Data Types: `uint8`

Examples**Specify XCP Memory Configuration Parameters**

For an example that uses this class, see “Customize Connectivity for XCP External Mode Simulations”.

Version History

Introduced in R2021a

See Also

`target.create` | `target.XCPTCPIPTransport` | `target.XCPSerialTransport`

Topics

“Customize Connectivity for XCP External Mode Simulations”

“Memory Allocation for Communication Buffers During XCP External Mode Simulation”

External Websites

<https://www.asam.net/standards/detail/mcd-1-xcp/wiki/>

target.XCPSerialTransport Class

Namespace: target

Represent XCP serial transport protocol layer

Description

Use the `target.XCPSerialTransport` class, which inherits functionality from `target.XCPTransport`, to represent the XCP serial transport protocol layer for your target hardware.

Examples

XCP Serial Transport

This code snippet shows how you can use the `target.XCPSerialTransport` class to represent the XCP serial transport protocol layer for your target hardware.

```
xcpSerialTransport = ...  
    target.create('XCPSerialTransport', ...  
        'Name', 'XCP Serial Transport');
```

Version History

Introduced in R2021a

See Also

`target.create` | `target.XCP` | `target.XCPTransport`

Topics

“Customize Connectivity for XCP External Mode Simulations”

updateFilePathsAndExtensions

Update files in 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 `myBuildInfo` with missing paths or file extensions.

```
myBuildInfo = RTW.BuildInfo;
addSourcePaths(myBuildInfo,fullfile(pwd, ...
    'etcproj','/etc'),'test');
addSourceFiles(myBuildInfo,{'etc' 'test1' ...
    'test2'},'', 'test');
before = getSourceFiles(myBuildInfo,true,true);
```

```
>> before
```

```
before =
```

```
    'etc'    '\test1'    '\test2'
```

```
updateFilePathsAndExtensions(myBuildInfo);
after = getSourceFiles(myBuildInfo,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 — Build information object
object

RTW.BuildInfo object that contains information for compiling and linking generated code.

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'

Version History

Introduced in R2006a

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFileSeparator`

Topics

"Customize Post-Code-Generation Build Processing"

updateFileSeparator

Update file separator character for file lists in build information

Syntax

```
updateFileSeparator(buildinfo,separator)
```

Description

`updateFileSeparator(buildinfo,separator)` changes instances of the current file separator (/ or \) in the 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. If the `GenerateMakefile` parameter is set, the code generator overrides the default separator and updates the build information after evaluating the `PostCodeGenCommand` configuration parameter.

Examples

Update File Separator in Build Information

Update object `myBuildInfo` to apply the Windows file separator.

```
myBuildInfo = RTW.BuildInfo;
updateFileSeparator(myBuildInfo, '\');
```

Input Arguments

buildinfo — `RTW.BuildInfo` object

object

`RTW.BuildInfo` object that contains information for compiling and linking generated code.

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: '\'

Version History

Introduced in R2006a

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFilePathsAndExtensions`

Topics

“Customize Post-Code-Generation Build Processing”

Embedded Coder Blocks

Byte Pack

Convert input signals to uint8 vector



Libraries:

Embedded Coder / Embedded Targets / Host Communication

Description

The Byte Pack block receives input signals of one or more data types and converts the data to one uint8 vector for output. Use block parameters to specify data types of the input signals and the alignment of the data in the vector that the block outputs. Because the UDP protocol transmits data in uint8 format, you can use this block to reformat data for UDP transmission by connecting the output of this block to the input of a UDP Send block.

Ports

Input

Port_1 — Signals to convert

signal of type double (default) | array of signal data

Array of input signals of one or more data types.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Converted signal data

vector

Converted block input, returned as a vector of uint8 data.

Data Types: uint8

Parameters

Input port data types (cell array) — Data types of block input signals

{'double'} (default) | cell array of Simulink data types

Specify the Simulink data types of input signals that the block receives in a cell array. In the cell array, specify data types in the order in which the block input port receives the signal data. For example, if the block receives data in the order uint32, uint32, uint16, double, uint8, double, and single, specify this cell array:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

The block then provides the number of block inputs.

Byte alignment — Byte boundary for data type alignment

1 (default) | 2 | 4 | 8

Specify how to align the data types of input data to form the `uint8` vector output in bytes. Alignment can occur on 1, 2, 4, or 8-byte boundaries. Based on the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithms for each element in the output vector begin on a byte boundary specified by the alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

To get the tightest packing without holes between data types in the various combinations of data types, select 1.

You can have multiple data types of varying lengths. In such cases, 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.

For example, assume that you specify this cell array for **Input port data types (cell array)**:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

Assume that you set **Byte alignment** to 2. Each new value begins 2 bytes from the previous data boundary. When the signals are scalar values, the:

- First signal value in the vector starts at 0 bytes.
- Second signal value starts at 2 bytes.
- Third signal value starts at 4 bytes.
- Fourth signal value starts at 6 bytes.
- Fifth signal value starts at 8 bytes.
- Sixth signal value starts at 10 bytes
- Seventh signal value starts at 12 bytes.

The packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

Version History

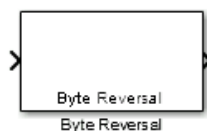
Introduced in R2011a

See Also

Byte Reversal | Byte Unpack | UDP Send

Byte Reversal

Reverse order of bytes in input word



Libraries:

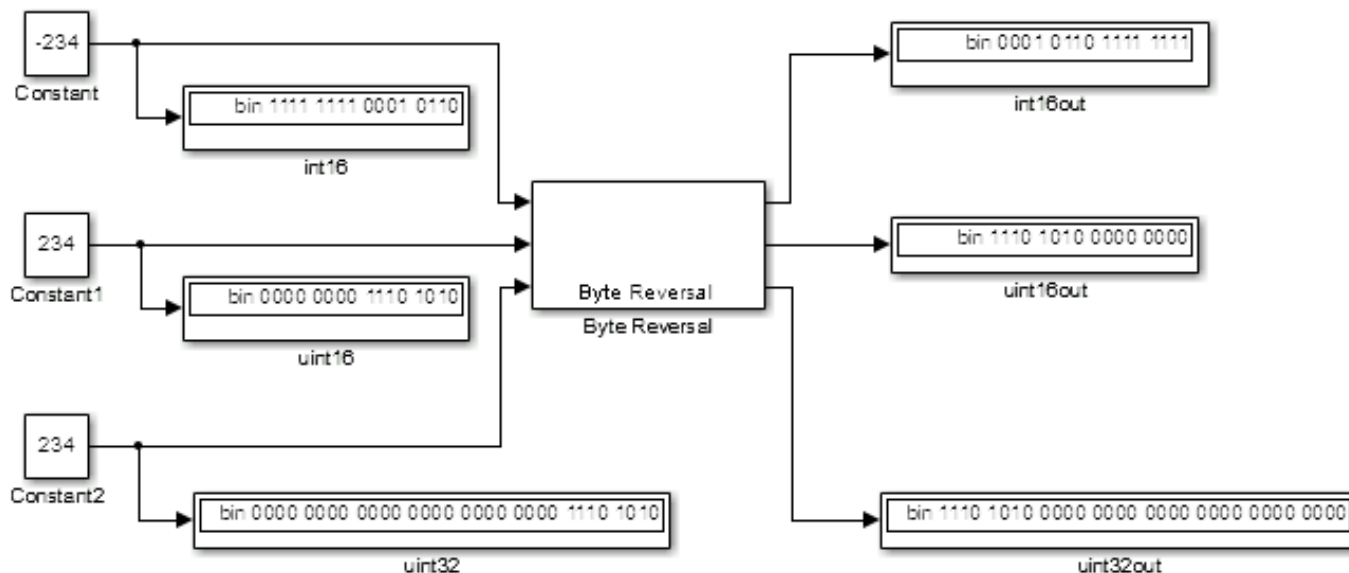
Embedded Coder / Embedded Targets / Host Communication

Description

The Byte Reversal block changes the order of the bytes in data that you input to the block. Use this block when a process communicates between target computers that use different endianness, such as between Intel® processors that are little endian and other processors that are big endian. Texas Instruments® processors are little-endian by default.

To exchange data between processors that have different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

This model shows byte reversal for three inputs. The input and output ports match for each path.



Ports

Input

Port_1 — Data to convert
scalar, vector, matrix

Data for which the block changes the byte ordering. The number of input ports used by the block depends on the value that you specify for the **Number of inputs** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed_point` | `enumerated` | `bus`

Output

Port_1 — Converted data
scalar, vector, matrix

Data with modified byte ordering. The number of output ports used by the block depends on the value that you specify for the **Number of inputs** parameter. Each input port maps to the matching output port. Data received on input port 1 is sent through output port 1, and so on.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed_point` | `enumerated` | `bus`

Parameters

Number of inputs — Number of block inputs
1 (default) | integer

Specify the number of block inputs. The block creates that number of input ports and output ports. Each input port maps to the matching output port. Data received on input `Port_1` is sent through output `Port_1`, and so on.

Reversing the byte ordering does not change the data type. Input and output retain matching data types.

Version History

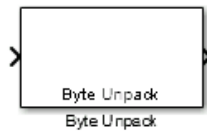
Introduced in R2011a

See Also

Byte Pack | Byte Unpack

Byte Unpack

Convert `uint8` vector to input signals

**Libraries:**

Embedded Coder / Embedded Targets / Host Communication

Description

The Byte Unpack block receives a `uint8` vector and converts the vector to output signals of different Simulink data types based on content of the input vector. You can use block parameters to specify dimensions and data types of the output signals and the alignment of the data in the individual vectors that the block outputs. Because the UDP protocol transmits data in `uint8` format, you can use this block to reformat data that is received as a UDP packet for use in a model by connecting the input of this block to the output of a UDP Send block.

Ports

Input

Port_1 — Signal to convert
vector

Input vector of `uint8`.

Data Types: `uint8`

Output

Port_1 — Converted signal data
signal of type `double` (default) | array of signal data

Array of input signals of one or more data types.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Output port dimensions (cell array) — Dimensions of block output signals
{[1]} (default) | cell array of data dimension specifications

Specify the dimensions of the output signals that the block outputs in a cell array. Each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Specify dimensions that align with data converted by the corresponding Byte Pack block in the model.

For example, assume the corresponding Byte Pack block specifies these input port data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```


To specify scalar and matrix output, you might set the **Output port dimensions (cell array)** parameter to:

```
{1,1,[2,4],[4,4],[2,2],1,[3,3]}
```

To apply the same dimension across the output signals, you can specify one dimension value.

Output port data types (cell array) — Data types of block output signals

{'double'} (default) | cell array of Simulink data types

Specify the Simulink data types of the individual input signals received by the corresponding Byte Pack block in the model in a cell array.

For example, if the corresponding Byte Pack block specifies these input port data types, specify the same cell array for this parameter.

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

To apply the same data type to each output signal, you need to specify the data type only once.

Byte Alignment — Byte boundary for data type alignment

1 (default) | 2 | 4 | 8

Specify how to align the data types of output data to form the `uint8` vector input in bytes. Specify the alignment value that matches the value specified for the corresponding Byte Pack block in the model.

For example, if the corresponding Byte Pack block sets the byte alignment to 2, set this parameter to 2.

Version History

Introduced in R2011a

See Also

Byte Pack | Byte Reversal | UDP Receive

CAN Pack

Pack individual signals into CAN message



Libraries:

Vehicle Network Toolbox / CAN Communication
C2000 Microcontroller Blockset / Target Communication
Embedded Coder / Embedded Targets / Host Communication
Embedded Coder Support Package for Infineon AURIX TC3x
Microcontrollers / Utilities
Embedded Coder Support Package for Infineon AURIX TC4x
Microcontrollers / Utilities
Simulink Real-Time / CAN / CAN MSG blocks

Description

The CAN Pack block loads signal data into a CAN message at specified intervals during the simulation.

The CAN Pack block supports:

- Simulink accelerator and rapid accelerator modes. You can speed up the execution of Simulink models. For more information, see “Acceleration”.
- Model referencing. Your model can include other Simulink models as modular components. For more information, see “Model References”.

Tip

- This block can be used to encode the signals of J1939 parameter groups up to 8 bytes. However, to work with J1939 messages, it is preferable to use the blocks in the J1939 Communication block library instead of this block.
-

Ports

Input

Data — CAN message signal input

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | boolean

The 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 message has four signals, the block can have four input ports.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Tip Even if you have a zero-length message with no signals, you still must provide dummy data to the input port, for purposes of defining the sample time. In this case, the data value itself is ignored.

CAN Msg ID — CAN message identifier

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

The CAN Msg ID input port allows you to dynamically specify the CAN message identifier. This port exists only when the **“Specify CAN identifier from input port” on page 3-0** parameter is selected.

Output**CAN Msg** — CAN message output

CAN_MESSAGE | CAN_MESSAGE_BUS

This block has one output port, CAN Msg. The CAN Pack block takes the specified input signals and packs them into a CAN message. The output data type is determined by the **Output as bus** parameter setting.

Parameters**Data input as** — Select your data signal

raw data (default) | manually specified signals | CANdb specified signals | ARXML specified signals

- **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.

The conversion formula is:

$$\text{raw\_value} = (\text{physical\_value} - \text{Offset}) / \text{Factor}$$

where **physical_value** is the original value of the signal and **raw_value** is the packed signal value.

- **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.
- **ARXML specified signals:** Allows you to specify an ARXML file for signal definitions. This disables some remote options. After reading data from the ARXML file, you can switch to **manually specified signals** to further modify the table.

Programmatic Use**Block Parameter:** DataFormat**Type:** string | character vector**Values:** 'raw data' | 'manually specified signals' | 'CANdb specified signals' | 'ARXML specified signals'**Default:** 'raw data'**Database file** — ARXML or CAN database file

character vector

This option is available if you specify the **Data is input as** parameter as **CANdb specified signals** or **ARXML specified signals**. Click **Browse** to find the database file on your system.

The message list specified in the database file populates the **Message** section of the dialog box. The database file also populates the **Signals** table for the selected message.

File names that contain non-alphanumeric characters such as equal signs, ampersands, and so on are not valid database file names. You can use periods in your database name. Before you use the database files, rename them with non-alphanumeric characters.

Programmatic Use

Block Parameter: CANdbFile

Type: string | character vector

Message list — CAN message list

array of character vectors

This option is available if you specify database in the **Data is input as** field and you specify a database file in the **Database file** field. Select the message to display signal details in the **Signals** table.

Programmatic Use

Block Parameter: MsgList

Type: string | character vector

Name — CAN message name

CAN Msg (default) | character vector

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 is not available if you choose to use signals from a database file.

Programmatic Use

Block Parameter: MsgName

Type: string | character vector

Identifier type — CAN identifier type

Standard (11-bit identifier) (default) | Extended (29-bit identifier)

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 database specified signals, the **Identifier type** inherits the type from the database.

Programmatic Use

Block Parameter: MsgIDType

Type: string | character vector

Values: 'Standard (11-bit identifier)' | 'Extended (29-bit identifier)'

Default: 'Standard (11-bit identifier)'

CAN Identifier — CAN message ID

0 (default) | 0 to 536870911

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 by using the hex2dec function. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use**Block Parameter:** MsgIdentifier**Type:** string | character vector**Values:** '0' to '536870911'**Specify CAN identifier from input port** — CAN message ID value

off (default) | on

When selected, you can dynamically control the CAN identifier (CAN ID) by specifying the value through an input port on the block. Selecting this option adds an input port named “CAN Msg ID” on page 3-0 .

Programmatic Use**Block Parameter:** CANIDInput**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Length (bytes)** — CAN message length

8 (default) | 0 to 8

Specify the length of your CAN message from 0 to 8 bytes. If you are using database specified signals for your data input, the database 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.

Programmatic Use**Block Parameter:** MsgLength**Type:** string | character vector**Values:** '0' to '8'**Default:** '8'**Remote frame** — CAN message as remote frame

off (default) | on

Specify the CAN message as a remote frame.

Programmatic Use**Block Parameter:** Remote**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Output as bus** — CAN message as bus

off (default) | on

Select this option for the block to output CAN messages as a Simulink bus signal. For more information on Simulink bus objects, see “Composite Interfaces”.

Programmatic Use**Block Parameter:** BusOutput**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Add signal** — Add CAN signal

Add a new signal to the signal table.

Programmatic Use

None

Delete signal — Remove CAN signal

Remove the selected signal from the signal table.

Programmatic Use

None

Signals — Signals table
table

This table appears if you choose to specify signals manually or define signals by using a database file.

If you are using a database file, the data in the file populates this table 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 in this table. Each signal that you create has these 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 these options:

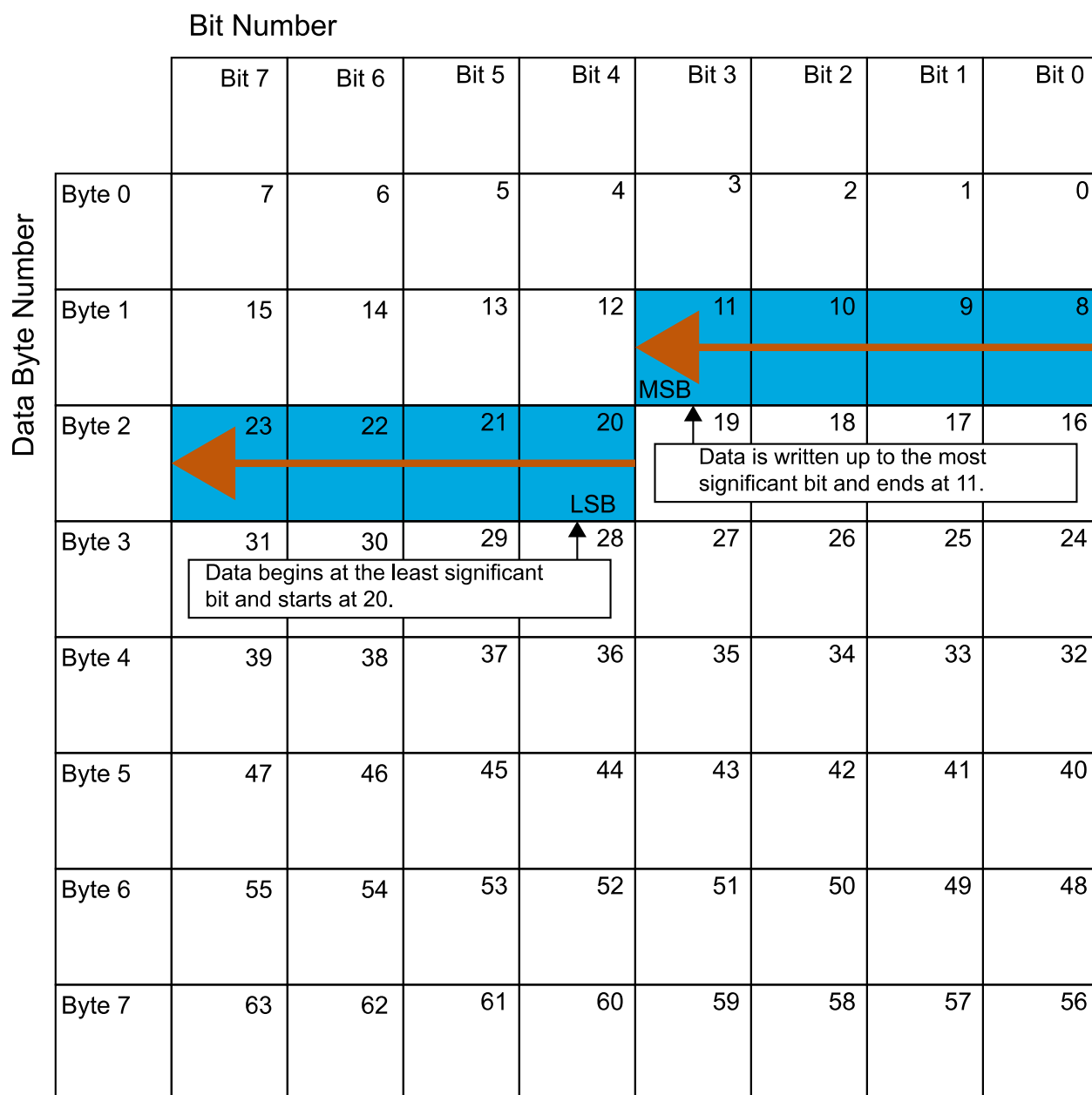
- **LE:** Where the byte order is in little-endian format (Intel). In this format you count bits from the least significant bit, to the most significant bit. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.

		Bit Number							
		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number	Byte 0	7	6	5	4	3	2	1	0
	Byte 1	15	14	13	12	11	10	9	8
	Byte 2	23	22	21	20	19	18	17	16
	Byte 3	31	30	29	28	27	26	25	24
	Byte 4	39	38	37	36	35	34	33	32
	Byte 5	47	46	45	44	43	42	41	40
	Byte 6	55	54	53	52	51	50	49	48
	Byte 7	63	62	61	60	59	58	57	56

Diagram illustrating Little-Endian Byte Order. The data is counted from the Least-Significant Bit (LSB) to the Most-Significant Bit (MSB). The data begins at the least significant bit and starts at 20. The data is written up to the most significant bit and ends at 27.

Address

- BE: Where byte order is in big-endian format (Motorola®). In this format you count bits from 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.



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 time step:

- **Standard**: The signal is packed at each time step.
- **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 these signals with the following types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block packs Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block packs Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that time step.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be **Multiplexed**. The value you provide 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. For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. For more information, see the **Data input as** parameter conversion formula.

Min, Max

Define a range of signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For database specified signals, these settings are read from the 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 the settings.

Programmatic Use

Block Parameter: `SignalInfo`

Type: `string` | `character vector`

Version History

Introduced in R2009a

R2025a: Updated parameters: ARXML support

To support ARXML file definitions, the following parameters are updated:

- **Data input as** — Has a new option `ARXML specified signals`.
- **CANdb file** — Has been renamed to **Database file** in the block mask. Its programmatic name remains `CANdbFile`.

R2024b: New parameter: Specify CAN identifier from input port

The parameter **Specify CAN identifier from input port** adds an input port **CAN Msg ID**, to support dynamic specifying of CAN identifier.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

CAN Unpack

Topics

“Supported Block Features” (Vehicle Network Toolbox)

“Design Your Model for Effective Acceleration”

CAN Unpack

Unpack individual signals from CAN messages



Libraries:

Vehicle Network Toolbox / CAN Communication
C2000 Microcontroller Blockset / Target Communication
Embedded Coder / Embedded Targets / Host Communication
Embedded Coder Support Package for Infineon AURIX TC3x
Microcontrollers / Utilities
Embedded Coder Support Package for Infineon AURIX TC4x
Microcontrollers / Utilities
Simulink Real-Time / CAN / CAN MSG blocks

Description

The CAN Unpack block unpacks a CAN message into signal data using the specified output parameters at every time step. Data is output as individual signals.

The CAN Unpack block supports:

- The use of Simulink accelerator and rapid accelerator modes. Using this feature, you can speed up the execution of Simulink models. For more information, see “Acceleration”.
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components. For more information, see “Model References”.

Tip

- To process every message coming through a channel, do not use the CAN Unpack block in a function trigger subsystem. For more information, see Speedgoat® documentation.
 - This block can be used to decode the signals of J1939 parameter groups up to 8 bytes. However, to work with J1939 messages, it is preferable to use the blocks in the J1939 Communication block library instead of this block.
-

Ports

Input

CAN Msg — CAN message input
CAN_MESSAGE | CAN_MESSAGE_BUS

This block has one input port, **CAN Msg**. The block takes the specified input CAN messages and unpacks their signal data to separate outputs.

The block supports the following signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Output

Data — CAN signal output

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | boolean

The block has one output port by default. The number of output ports is dynamic and depends on the number of signals that you specify for the block to output. For example, if your message has four signals, the block can have four output ports.

For signals specified manually or by a database, the default output data type for CAN signals 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.

Additional output ports can be added by selecting the options in the parameters **Output ports** pane. For more information, see the parameters **Output identifier**, **Output timestamp**, **Output error**, **Output remote**, **Output length**, and **Output status**.

Parameters

Data to output as — Select your data signal

raw data (default) | manually specify signals | CANdb specified signals | ARXML specified signals

- **raw data:** Output data as a uint8 vector array. If you select this option, you specify only the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.

The conversion formula is:

$$\text{physical\_value} = \text{raw\_value} * \text{Factor} + \text{Offset}$$

where **raw_value** is the unpacked signal value and **physical_value** is the scaled signal value.

- **manually specified signals:** You can 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 that you specify. For example, if you specify four signals, your block has four output ports.
- **CANdb specified signals:** You can 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.
- **ARXML specified signals:** Allows you to specify an ARXML file for signal definitions. This disables some remote options. After reading data from the ARXML file, you can switch to **manually specified signals** to further modify the table.

Programmatic Use

Block Parameter: DataFormat

Type: string | character vector

Values: 'raw data' | 'manually specified signals' | 'CANdb specified signals' | 'ARXML specified signals'

Default: 'raw data'

Database file — ARXML or CAN database file

character vector

This option is available if you specify the **Data to output as** parameter as **CANdb specified signals** or **ARXML specified signals**. Click **Browse** to find the database file on your system. The messages and signal definitions specified in the database file populate the **Message** section of the dialog box. The signals specified in the database file populate **Signals** table. File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid database file names. You can use periods in your database name. Rename database files with non-alphanumeric characters before you use them.

Programmatic Use**Block Parameter:** CANdbFile**Type:** string | character vector**Message list** — CAN message list

array of character vectors

This option is available if you specify a database for the **Data is input as** field, and you specify a database file in the **Database file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

Programmatic Use**Block Parameter:** MsgList**Type:** string | character vector**Name** — CAN message name

CAN Msg (default) | character vector

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.

Programmatic Use**Block Parameter:** MsgName**Type:** string | character vector**Identifier type** — CAN identifier type

Standard (11-bit identifier) (default) | Extended (29-bit identifier)

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 database specified signals, the **Identifier type** inherits the type from the database.

Programmatic Use**Block Parameter:** MsgIDType**Type:** string | character vector**Values:** 'Standard (11-bit identifier)' | 'Extended (29-bit identifier)'**Default:** 'Standard (11-bit identifier)'**CAN Identifier** — CAN message identifier

0 (default) | 0 to 536870911

Specify your CAN message ID. This number must be an 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 by using the `hex2dec` function. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use

Block Parameter: `MsgIdentifier`

Type: `string` | `character vector`

Values: '0' to '536870911'

Length (bytes) — CAN message length

8 (default) | 0 to 8

Specify the length of your CAN message from 0 to 8 bytes. If you are using database specified signals for your output data, the database file defines the length of your message. Otherwise, this field defaults to 8. This option is available if you choose to output raw data or manually specify signals.

Note: When a frame is received whose length is different from the value specified in the mask or network description file, that frame is discarded.

Programmatic Use

Block Parameter: `MsgLength`

Type: `string` | `character vector`

Values: '0' to '8'

Default: '8'

Add signal — Add CAN signal

Add a signal to the signal table.

Programmatic Use

None

Delete signal — Remove CAN signal

Remove the selected signal from the signal table.

Programmatic Use

None

Signals — Signals table

table

If you choose to specify signals manually or define signals by using a database file, this table appears.

If you are using a database file, the data in the file populates this table and you cannot edit the fields. To edit signal information, switch to specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal that you create has these 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 these options:

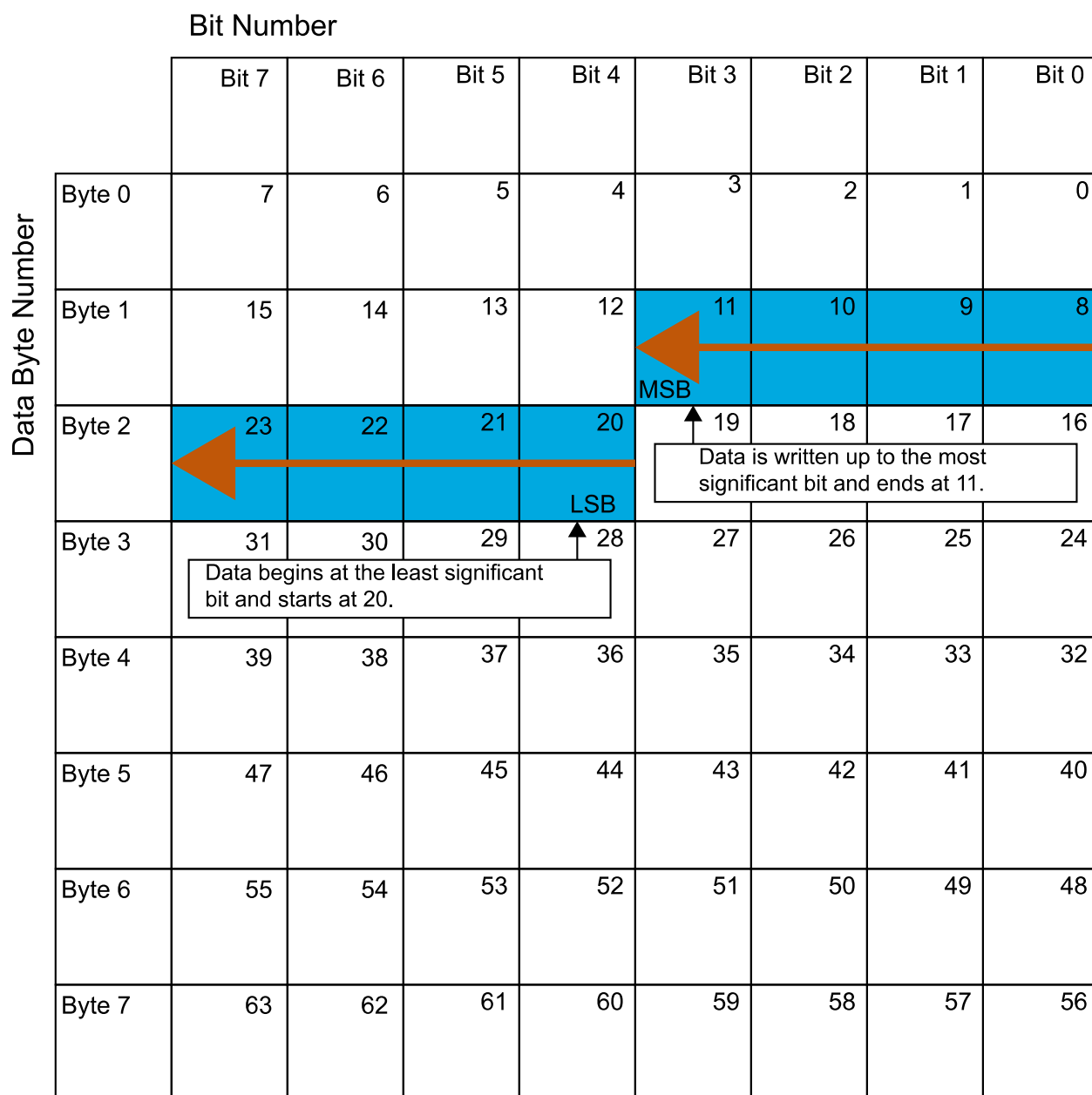
- LE: Where the byte order is in little-endian format (Intel). In this format you count bits from the least-significant bit to the most-significant bit. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.

		Bit Number							
		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number	Byte 0	7	6	5	4	3	2	1	0
	Byte 1	15	14	13	12	11	10	9	8
	Byte 2	23	22	21	20	19	18	17	16
	Byte 3	31	30	29	28	27	26	25	24
	Byte 4	39	38	37	36	35	34	33	32
	Byte 5	47	46	45	44	43	42	41	40
	Byte 6	55	54	53	52	51	50	49	48
	Byte 7	63	62	61	60	59	58	57	56

Diagram illustrating Little-Endian Byte Order. The data is counted from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). The data begins at the least significant bit and starts at 20. The data is written up to the most significant bit and ends at 27.

Address

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from 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.



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 time step:

- **Standard**: The signal is unpacked at each time step.
- **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 these values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block unpacks Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block unpacks Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that time step.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be **Multiplexed**. The value you provide 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). For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. For more information, see the **Data input as** parameter conversion formula.

Min, Max

Define a range of raw signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For database specified signals, these settings are read from the 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.

Programmatic Use

Block Parameter: `SignalInfo`

Type: `string` | `character vector`

Output identifier — Add CAN ID output port

`off` (default)

Select this option to output a CAN message identifier. The data type of this port is `uint32`.

Programmatic Use

Block Parameter: `IDPort`

Type: `string` | `character vector`

Values: `'off'` | `'on'`

Default: `'off'`

Output timestamp — Add Timestamp output port

`off` (default) | `on`

Select this option to output the message timestamp. This value indicates when the message was received, measured as the number of seconds elapsed since the model simulation began. This option adds a new output port to the block. The data type of this port is `double`.

Programmatic Use

Block Parameter: `TimestampPort`

Type: `string` | `character vector`

Values: `'off'` | `'on'`

Default: `'off'`

Output error — Add Error output port

`off` (default) | `on`

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. If the output value is 0, there is no error. The data type of this port is `uint8`.

Programmatic Use

Block Parameter: `ErrorPort`

Type: `string` | `character vector`

Values: `'off'` | `'on'`

Default: `'off'`

Output remote — Add Remote output port

`off` (default) | `on`

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`.

Programmatic Use

Block Parameter: `RemotePort`

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Output length — Add Length output port

off (default) | on

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`.

Programmatic Use

Block Parameter: LengthPort

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Output status — Add Status output port

off (default) | on

Select this option to output the message received status. The status is 1 if the block receives a 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`.

Programmatic Use

Block Parameter: StatusPort

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Version History

Introduced in R2009a

R2025a: Updated parameters: ARXML support

To support ARXML file definitions, the following parameters are updated:

- **Data to output as** — Has a new option `ARXML` specified signals.
- **CANdb file** — Has been renamed to **Database file** in the block mask. Its programmatic name remains `CANdbFile`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

CAN Pack

Topics

“Supported Block Features” (Vehicle Network Toolbox)

“Design Your Model for Effective Acceleration”

CAN FD Pack

Pack individual signals into message for CAN FD bus



Libraries:

Vehicle Network Toolbox / CAN FD Communication
C2000 Microcontroller Blockset / Target Communication
Embedded Coder Support Package for Infineon AURIX TC3x
Microcontrollers / Utilities
Embedded Coder Support Package for Infineon AURIX TC4x
Microcontrollers / Utilities
Simulink Real-Time / CAN / CAN-FD MSG blocks

Description

The CAN FD Pack block loads signal data into a message at specified intervals during the simulation.

To use this block, you also need a license for Simulink software.

The CAN FD Pack block supports:

- The use of Simulink accelerator mode. Using this feature, you can speed up the execution of Simulink models. For more information, see “Acceleration”.

Tip

- To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block.
-

Ports

Input

Data — CAN FD message signal input

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | boolean

The CAN FD Pack block has one input port by default. The number of block inputs is dynamic and depends on the number of signals that you specify for the block. For example, if your message has four signals, the block can have four input ports.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Tip Even if you have a zero-length message with no signals, you still must provide dummy data to the input port, for purposes of defining the sample time. In this case, the data value itself is ignored.

CAN Msg ID — CAN message identifier

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

The CAN Msg ID input port allows you to dynamically specify the CAN message identifier. This port exists only when the “**Specify CAN identifier from input port**” on page 3-0 parameter is selected.

Output

Msg — CAN FD message output
CAN_FD_MESSAGE_BUS

This block has one output port, Msg. The CAN FD Pack block takes the specified input signals and packs them into a CAN FD message, output as a Simulink CAN_FD_MESSAGE_BUS signal. For more information on Simulink bus objects, see “Composite Interfaces”.

Parameters

Data input as — Select your data signal

raw data (default) | manually specified signals | CANdb specified signals | ARXML specified signals

- **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.

The conversion formula is:

$$\text{raw\_value} = (\text{physical\_value} - \text{Offset}) / \text{Factor}$$

where **physical_value** is the original value of the signal and **raw_value** is the packed signal value.

- **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.
- **ARXML specified signals**: Allows you to specify an ARXML file for signal definitions. This disables some remote options. After reading data from the ARXML file, you can switch to manually specified signals to further modify the table.

Programmatic Use

Block Parameter: DataFormat

Type: string | character vector

Values: 'raw data' | 'manually specified signals' | 'CANdb specified signals' | 'ARXML specified signals'

Default: 'raw data'

Database file — ARXML or CAN database file

character vector

This option is available if you specify the **Data is input as** parameter as **CANdb specified signals** or **ARXML specified signals**. Click **Browse** to find the database file on your system. The message list specified in the database file populates the **Message** section of the dialog box. The database file also populates the **Signals** table for the selected message. File names that contain non-

alphanumeric characters such as equal signs, ampersands, and so on are not valid database file names. You can use periods in your database name. Before you use the database files, rename them with non-alphanumeric characters.

Programmatic Use**Block Parameter:** CANdbFile**Type:** string | character vector**Message list** — CAN message list

array of character vectors

This option is available if you specify a database in the **Data is input as** field and you specify a database file in the **Database file** field. Select the message to display signal details in the **Signals** table.

Programmatic Use**Block Parameter:** MsgList**Type:** string | character vector**Name** — CAN FD message name

CAN Msg (default) | character vector

Specify a name for your CAN FD message. The default is CAN Msg. This option is available if you choose to input raw data or manually specify signals. This option is not available if you choose to use signals from a database file.

Programmatic Use**Block Parameter:** MsgName**Type:** string | character vector**Protocol mode** — CAN FD message protocol

CAN FD (default) | CAN

Specify the message protocol mode.

Programmatic Use**Block Parameter:** ProtocolMode**Type:** string | character vector**Values:** 'CAN FD' | 'CAN'**Default:** 'CAN FD'**Identifier type** — CAN identifier type

Standard (11-bit identifier) (default) | Extended (29-bit identifier)

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 database specified signals, the **Identifier type** inherits the type from the database.

Programmatic Use**Block Parameter:** MsgIDType**Type:** string | character vector**Values:** 'Standard (11-bit identifier)' | 'Extended (29-bit identifier)'**Default:** 'Standard (11-bit identifier)'**Identifier** — Message identifier

0 (default) | 0 .. 536870911

Specify your 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 by using the hex2dec function. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use

Block Parameter: MsgIdentifier

Type: string | character vector

Values: '0' to '536870911'

Specify CAN identifier from input port — CAN message ID value

off (default) | on

When selected, you can dynamically control the CAN identifier (CAN ID) by specifying the value through an input port on the block. Selecting this option adds an input port named “CAN Msg ID” on page 3-0 .

Programmatic Use

Block Parameter: CANIDInput

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Length (bytes) — CAN FD message length

8 (default) | 0 to 64

Specify the length of your message. For CAN messages the value can be 0 to 8 bytes; for CAN FD the value can be 0 to 8, 12, 16, 20, 24, 32, 48, or 64 bytes. If you are using database specified signals for your data input, the database file defines the length of your message. This option is available if you choose to input raw data or manually specify signals.

Programmatic Use

Block Parameter: MsgLength

Type: string | character vector

Values: '0' to '8', '12', '16', '20', '24', '32', '48', '64'

Default: '8'

Remote frame — CAN message as remote frame

off (default) | on

(Disabled for CAN FD protocol mode.) Specify the CAN message as a remote frame.

Programmatic Use

Block Parameter: Remote

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Bit Rate Switch (BRS) — Enable bit rate switch

off (default) | on

(Disabled for CAN protocol mode.) Enable bit rate switch.

Programmatic Use

Block Parameter: BRSSwitch

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Add signal — Add CAN FD signal

Add a signal to the signal table.

Programmatic Use

None

Delete signal — Remove CAN FD signal

Remove the selected signal from the signal table.

Programmatic Use

None

Signals — Signals table table

This table appears if you choose to specify signals manually or define signals by using a database file.

If you are using a database file, the data in the file populates this table 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 in this table. Each signal that you create has these 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. For CAN the start bit must be an integer from 0 through 63, for CAN FD 0 through 511, within the number of bits in the message. (Note that message length is specified in bytes.)

Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64. The sum of all the signal lengths in a message is limited to the number of bits in the message length; that is, all signals must cumulatively fit within the length of the message. (Note that message length is specified in bytes and signal length in bits.)

Byte order

Select either of these options:

- **LE:** Where the byte order is in little-endian format (Intel). In this format you count bits from the least significant bit, to the most significant bit. 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.

		Bit Number							
		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number	Byte 0	7	6	5	4	3	2	1	0
	Byte 1	15	14	13	12	11	10	9	8
	Byte 2	23	22	21	20	19	18	17	16
	Byte 3	31	30	29	28	27	26	25	24
	Byte 4	39	38	37	36	35	34	33	32
	Byte 5	47	46	45	44	43	42	41	40
	Byte 6	55	54	53	52	51	50	49	48
	Byte 7	63	62	61	60	59	58	57	56

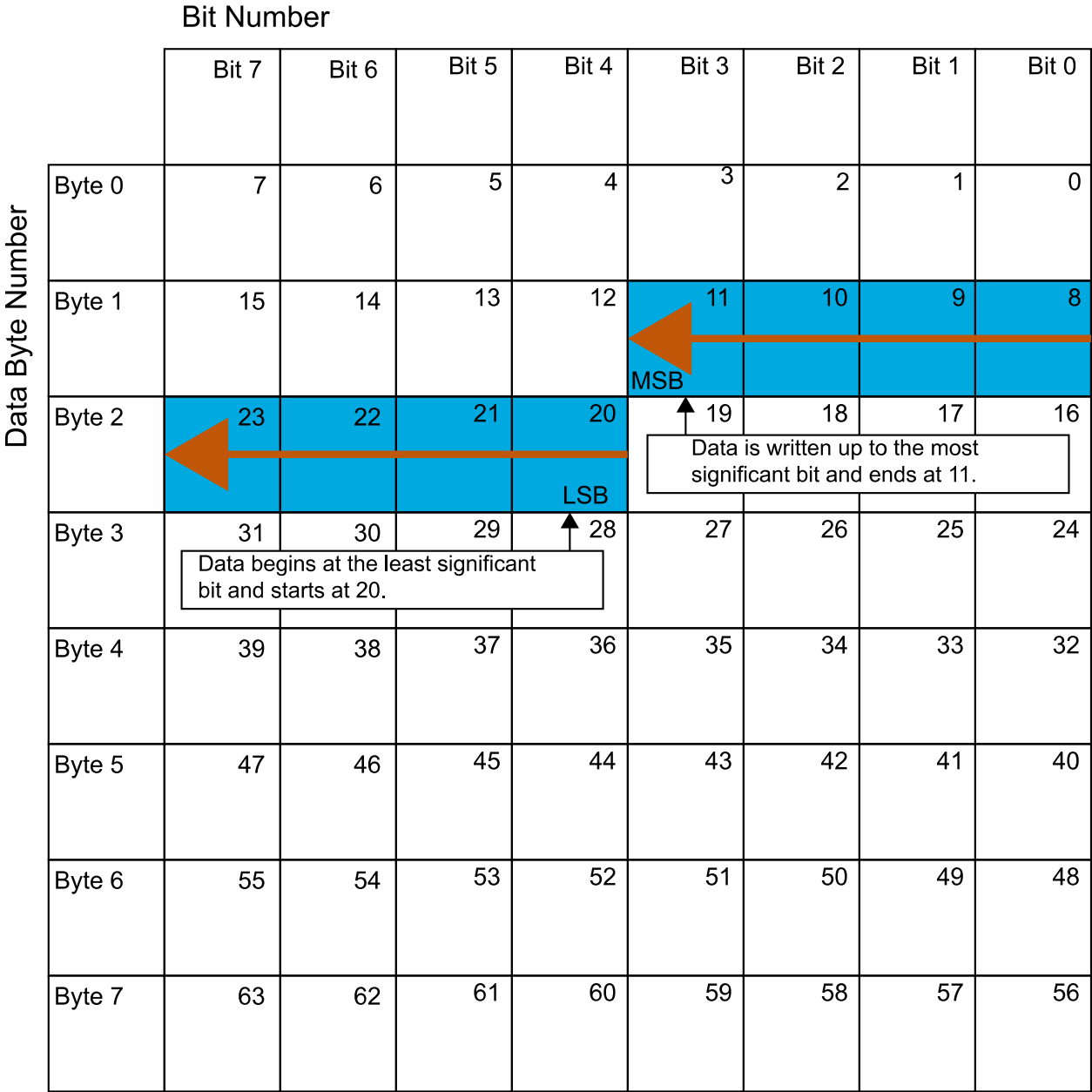
Diagram illustrating the bit numbering for CAN FD Pack data bytes (Byte 0 to Byte 7) in Little-Endian byte order. The diagram shows the bit positions (0 to 63) and the data flow direction (LSB to MSB).

Key annotations:

- LSB (Least Significant Bit):** Indicated by an arrow pointing to bit 20 in Byte 2.
- MSB (Most Significant Bit):** Indicated by an arrow pointing to bit 27 in Byte 3.
- Data flow:** Data begins at the least significant bit and starts at 20. Data is written up to the most significant bit and ends at 27.

Address

- BE: Where byte order is in big-endian format (Motorola). In this format you count bits from 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.



Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- signed (default)
- unsigned
- single
- double

Note: If you have a double signal that does not align exactly to the message byte boundaries, to generate code with Embedded Coder you must check **Support long long** under **Device Details** in the **Hardware Implementation** pane of the Configuration Parameters dialog.

Multiplex type

Specify how the block packs the signals into the message at each time step:

- **Standard**: The signal is packed at each time step.
- **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 these types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block packs Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block packs Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that time step.

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 the **Data input as** parameter conversion formula 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 the **Data input as** parameter conversion formula 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 database specified signals, these settings are read from the database file. 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.

Programmatic Use

Block Parameter: `SignalInfo`

Type: `string` | `character vector`

Version History

Introduced in R2018a**R2025a: Updated parameters: ARXML support**

To support ARXML file definitions, the following parameters are updated:

- **Data input as** — Has a new option `ARXML specified signals`.
- **CANdb file** — Has been renamed to **Database file** in the block mask. Its programmatic name remains `CANdbFile`.

R2024b: New parameter: Specify CAN identifier from input port

The parameter **Specify CAN identifier from input port** adds an input port **CAN Msg ID**, to support dynamic specifying of CAN identifier.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks**Topics**

“Supported Block Features” (Vehicle Network Toolbox)

“Design Your Model for Effective Acceleration”

“Composite Interfaces”

CAN FD Unpack

Unpack individual signals from CAN FD messages



Libraries:

Vehicle Network Toolbox / CAN FD Communication
C2000 Microcontroller Blockset / Target Communication
Embedded Coder Support Package for Infineon AURIX TC3x
Microcontrollers / Utilities
Embedded Coder Support Package for Infineon AURIX TC4x
Microcontrollers / Utilities
Simulink Real-Time / CAN / CAN-FD MSG blocks

Description

The CAN FD Unpack block unpacks a CAN FD message into signal data by using the specified output parameters at every time step. Data is output as individual signals.

To use this block, you also need a license for Simulink software.

The CAN FD Unpack block supports:

- Simulink accelerator mode. You can speed up the execution of Simulink models. For more information, see “Acceleration”.

Tip

- To process every message coming through a channel, do not use the CAN Unpack block in a function trigger subsystem. For more information, see Speedgoat documentation.
 - To work with J1939 messages, use the blocks in the J1939 Communication block library instead of this block.
-

Ports

Input

Msg — CAN FD message input
`CAN_FD_MESSAGE_BUS`

This block has one input port, Msg. The CAN FD Unpack block takes the specified input CAN messages and unpacks their signal data to separate outputs.

The block supports the following input signal data types: single, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. The block does not support fixed-point data types.

Output

Data — CAN message output
`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | boolean`

The CAN FD Unpack block has one output port by default. The number of data 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, labeled by signal name.

For manually or database 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.

Additional output ports can be added by the options in the parameters **Output ports** pane.

Parameters

Data to output as — Select your data signal

raw data (default) | manually specify signals | CANdb specified signals | ARXML specified signals

- **raw data:** Output data as a uint8 vector array. If you select this option, you specify only the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.

The conversion formula is:

$$\text{physical\_value} = \text{raw\_value} * \text{Factor} + \text{Offset}$$

where **raw_value** is the unpacked signal value and **physical_value** is the scaled signal value.

- **manually specified signals:** You can 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 that you specify. For example, if you specify four signals, your block has four output ports.
- **CANdb specified signals:** You can 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.
- **ARXML specified signals:** Allows you to specify an ARXML file for signal definitions. This disables some remote options. After reading data from the ARXML file, you can switch to manually specified signals to further modify the table.

Programmatic Use

Block Parameter: DataFormat

Type: string | character vector

Values: 'raw data' | 'manually specified signals' | 'CANdb specified signals' | 'ARXML specified signals'

Default: 'raw data'

Database file — ARXML or CAN database file

character vector

This option is available if you specify the **Data to output as** parameter as **CANdb specified signals** or **ARXML specified signals**. Click **Browse** to find the database file on your system. The messages and signal definitions specified in the database file populate the **Message** section of the dialog box. The signals specified in the database file populate the **Signals** table. File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth, are not valid

database file names. You can use periods in your database name. Rename database files with non-alphanumeric characters before you use them.

Programmatic Use**Block Parameter:** CANdbFile**Type:** string | character vector**Message list** — Message list

array of character vectors

This option is available if you specify a database for the **Data is input as** field, and you specify a database file in the **Database file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

Programmatic Use**Block Parameter:** MsgList**Type:** string | character vector**Name** — Message name

CAN Msg (default) | character vector

Specify a name for your message. The default is Msg. This option is available if you choose to output raw data or manually specify signals.

Programmatic Use**Block Parameter:** MsgName**Type:** string | character vector**Identifier type** — Identifier type

Standard (11-bit identifier) (default) | Extended (29-bit identifier)

Specify whether your 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 database specified signals, the **Identifier type** inherits the type from the database.

Programmatic Use**Block Parameter:** MsgIDType**Type:** string | character vector**Values:** 'Standard (11-bit identifier)' | 'Extended (29-bit identifier)'**Default:** 'Standard (11-bit identifier)'**Identifier** — Message identifier

0 (default) | 0 .. 536870911

Specify your message ID. This number must be an 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.

Programmatic Use**Block Parameter:** MsgIdentifier**Type:** string | character vector**Values:** '0' to '536870911'

Length (bytes) — CAN message length

8 (default) | 0 .. 8

Specify the length of your message. For CAN messages the value can be 0-8 bytes; for CAN FD the value can be 0-8, 12, 16, 20, 24, 32, 48, or 64 bytes. If you are using database specified signals for your output data, the database file defines the length of your message. This option is available if you choose to output raw data or manually specify signals.

Note: When a frame is received whose length is different from the value specified in the mask or network description file, that frame is discarded.

Programmatic Use**Block Parameter:** MsgLength**Type:** string | character vector**Values:** '0' to '8', '12', '16', '20', '24', '32', '48', '64'**Default:** '8'**Add signal** — Add CAN signal

Add a signal to the signal table.

Programmatic Use

None

Delete signal — Remove CAN signal

Remove the selected signal from the signal table.

Programmatic Use

None

Signals — Signals table

table

If you choose to specify signals manually or define signals by using a database file, this table appears.

If you are using a database file, the data in the file populates this table and you cannot edit the fields. To edit signal information, switch to specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal that you create has these 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. For CAN the start bit must be an integer from 0 through 63, for CAN FD 0 through 511, within the number of bits in the message. (Note that message length is specified in bytes.)

Length (bits)

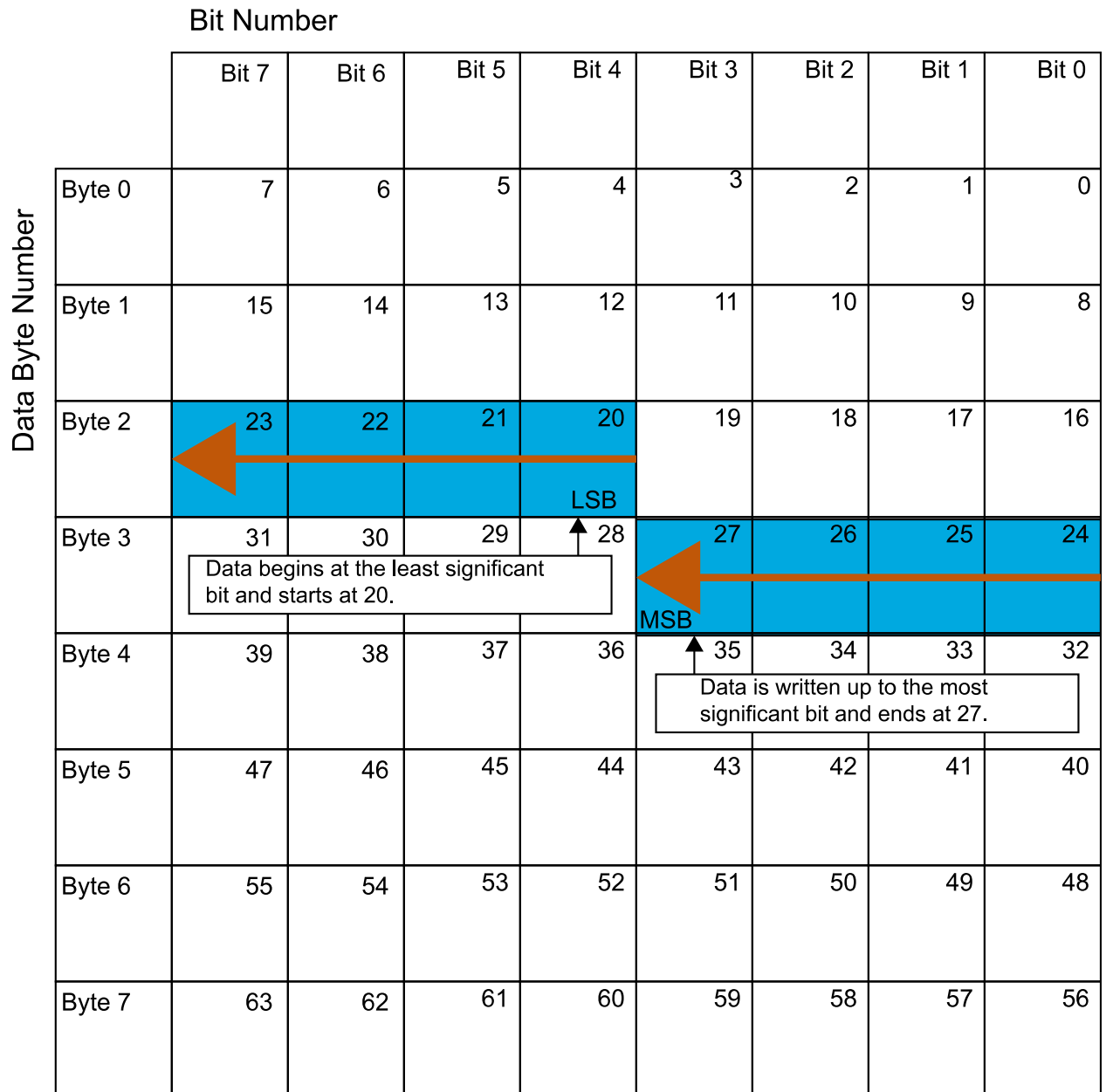
Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64. The sum of all the signal lengths in a message is limited to the number of bits

in the message length; that is, all signals must cumulatively fit within the length of the message. (Note that message length is specified in bytes and signal length in bits.)

Byte order

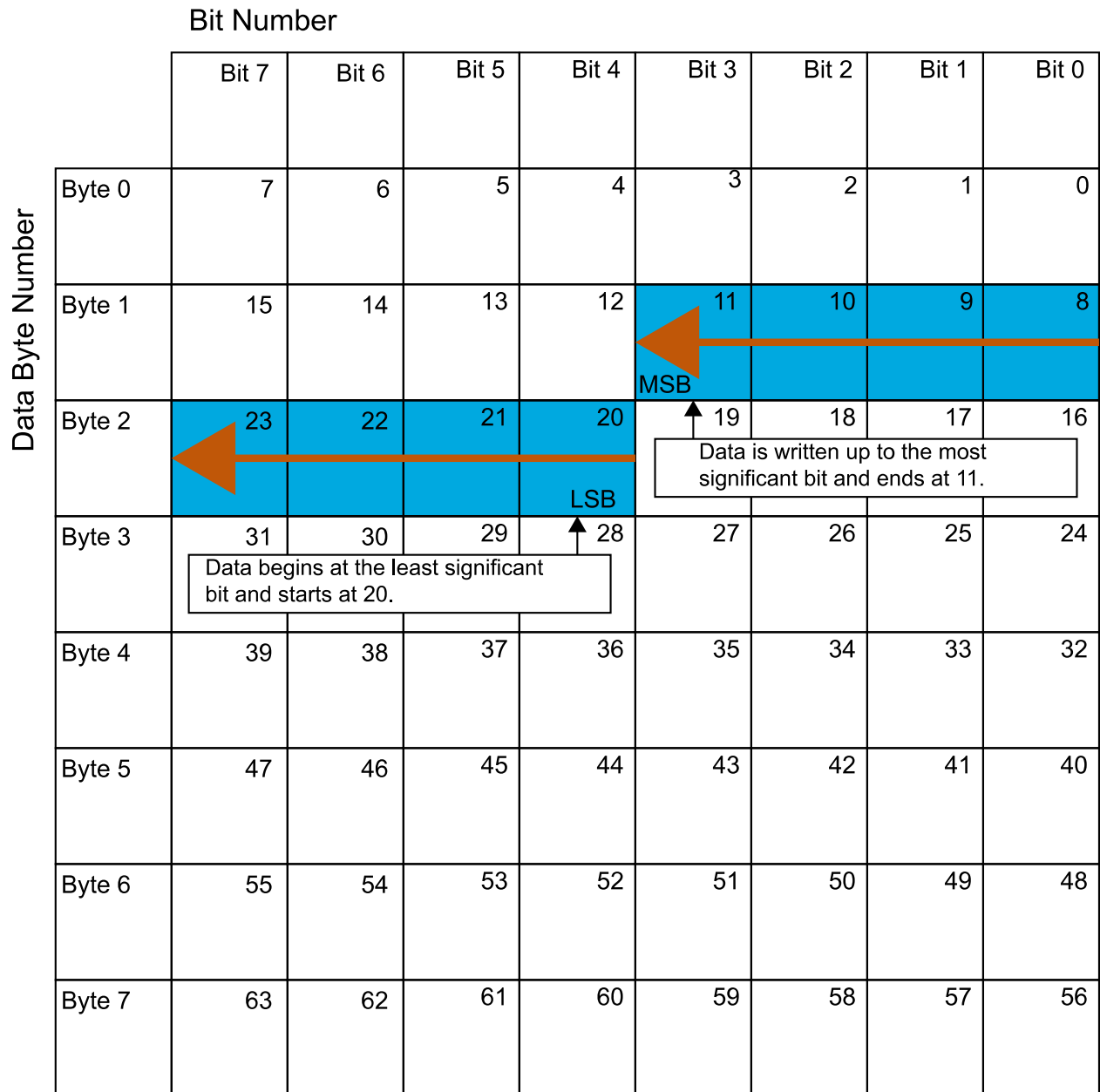
Select either of the following options:

- LE: Where the byte order is in little-endian format (Intel). In this format you count bits from the least-significant bit to the most-significant bit and proceeding to the next higher byte as you cross a byte boundary. 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.



Address

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the least-significant bit to the most-significant bit and proceeding to the next lower byte as you cross a byte boundary. 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.



Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- signed (default)
- unsigned
- single
- double

Note: If you have a **double** signal that does not align exactly to the message byte boundaries, to generate code with Embedded Coder you must check **Support long long** under **Device Details** in the **Hardware Implementation** pane of the Configuration Parameters dialog.

Multiplex type

Specify how the block unpacks the signals from the message at each time step:

- **Standard**: The signal is unpacked at each time step.
- **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 these values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	Not applicable
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	Not applicable

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every time step.
- If the value of Signal-D is 1 at a particular time step, then the block unpacks Signal-B along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is 0 at a particular time step, then the block unpacks Signal-C along with Signal-A and Signal-D in that time step.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that time step.

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). For more information, see the **Data input as** parameter conversion formula.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. For more information, see the **Data input as** parameter conversion formula.

Min, Max

Define a range of raw signal values. The default settings are `-Inf` (negative infinity) and `Inf`, respectively. For database specified signals, these settings are read from the 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.

Programmatic Use

Block Parameter: SignalInfo

Type: string | character vector

Output identifier — Add CAN ID output port

`off` (default) | `on`

Select this option to output a CAN message identifier. The data type of this port is `uint32`.

Programmatic Use

Block Parameter: IDPort

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Output timestamp — Add Timestamp output port

`off` (default) | `on`

Select this option to output the message timestamp. This value indicates when the message was received, measured as the number of seconds elapsed since the model simulation began. This option adds a new output port to the block. The data type of this port is `double`.

Programmatic Use

Block Parameter: TimestampPort

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Output error — Add Error output port

`off` (default) | `on`

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. If the output value is 0, there is no error. The data type of this port is `uint8`.

Programmatic Use

Block Parameter: ErrorPort

Type: string | character vector

Values: 'off' | 'on'

Default: 'off'

Output remote — Add Remote output port

`off` (default) | `on`

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`.

Programmatic Use**Block Parameter:** RemotePort**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Output length** — Add Length output port

off (default) | on

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`.

Programmatic Use**Block Parameter:** LengthPort**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Output status** — Add Status output port

off (default) | on

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`.

Programmatic Use**Block Parameter:** StatusPort**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Output Bit Rate Switch (BRS)** — Add BRS output port

off (default) | on

(Disabled for CAN protocol.) Select this option to output the message bit rate switch. This option adds a new output port to the block, which indicates if the CAN FD message bit rate switch is set. The data type of this port is `boolean`, indicating whether the bit rate for the data phase of the message is faster (`true`) or the same (`false`) as the bit rate of the arbitration phase.

For more information on BRS, see CAN FD - The basic idea.

Programmatic Use**Block Parameter:** BRSPort**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Output Error Status Indicator (ESI)** — Add ESI output port

off (default) | on

(Disabled for CAN protocol.) Select this option to output the message error status. This option adds a new output port to the block. The data type of this port is `boolean`, indicating if the CAN FD message error state indicator flag is set.

For more information on ESI, see CAN FD - The basic idea.

Programmatic Use**Block Parameter:** ESIPort**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'**Output Data Length Code (DLC)** — Add DLC output port

off (default) | on

(Disabled for CAN protocol.) Select this option to output the message data length. This option adds a new output port to the block. The data type of this port is double.

Programmatic Use**Block Parameter:** DLCPort**Type:** string | character vector**Values:** 'off' | 'on'**Default:** 'off'

Version History

Introduced in R2018a

R2025a: Updated parameters: ARXML support

To support ARXML file definitions, the following parameters are updated:

- **Data to output as** — Has a new option **ARXML specified signals**.
- **CANdb file** — Has been renamed to **Database file** in the block mask. Its programmatic name remains **CANdbFile**.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks**Topics**

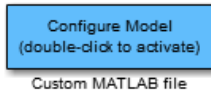
“Supported Block Features” (Vehicle Network Toolbox)

“Design Your Model for Effective Acceleration”

“Composite Interfaces”

Custom MATLAB file

Update active configuration parameters of parent model by using file containing custom MATLAB code



Libraries:

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/coder/simulinkcoder_core/rtwsampleconfig.m* with the Custom MATLAB file block. Adapt the script 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.

Version History

Introduced in R2011a

See Also

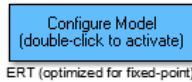
ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point) | ERT (optimized for fixed-point)

Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

ERT (optimized for fixed-point)

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



Libraries:

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.

Version History

Introduced in R2006b

See Also

ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point) | Custom MATLAB file

Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

ERT (optimized for floating-point)

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



Libraries:
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.

Version History

Introduced in R2006b

See Also

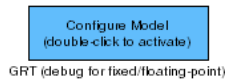
ERT (optimized for fixed-point) | GRT (debug for fixed/floating-point) | GRT (optimized for fixed/floating-point) | Custom MATLAB file

Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

GRT (debug for fixed/floating-point)

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



Libraries:
Embedded Coder / Configuration Wizards

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.

Version History

Introduced in R2006b

See Also

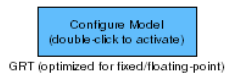
ERT (optimized for floating-point) | ERT (optimized for fixed-point) | GRT (optimized for fixed/floating-point) | Custom MATLAB file

Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

GRT (optimized for fixed/floating-point)

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



Libraries:
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.

Version History

Introduced in R2006b

See Also

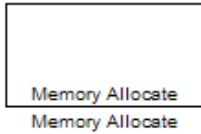
ERT (optimized for floating-point) | GRT (debug for fixed/floating-point) | ERT (optimized for fixed-point) | Custom MATLAB file

Topics

“Configure and Optimize Model with Configuration Wizard Blocks”

Memory Allocate

Allocate memory for new variable



Description

The Memory Allocate block, on C2xxx processors, directs the TI compiler to allocate a memory location for a new variable. Block parameters 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 parameter settings for the variable are valid, such as checking the variable name, data type, or section. You must check that the parameters settings are valid.

You do not connect the Memory Allocation block to other blocks in a model.

Parameters

Memory

Allocate memory for storing the variable. Specify the data type and size.

Variable name — Name for variable

myVariable (default) | string | character vector

Specify the name of the variable for which to allocate memory. The variable is allocated in the generated code.

Specify variable alignment — Variable alignment flag

off (default) | on

Select this parameter, if required by your target processor, to direct the compiler to align the new variable to a byte alignment boundary.

Parameter Dependencies

If you select this parameter, use parameter **Memory alignment boundary** to set the byte alignment boundary.

Memory alignment boundary — Memory alignment for variable

4 (default) | 1 | 2 | 8

Specify the alignment boundary for the variable data type in bytes. Alignment can occur on 1-, 2-, 4-, or 8-byte boundaries. If the variable contains multiple values, such as a vector or an array, the block aligns elements according to rules applied by the compiler.

Parameter Dependencies

To enable this parameter, select **Specify variable alignment**.

Data type — Data type for variable

uint32 (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | int64 | uint64 | boolean

Specify the data type for the variable.

Specify data type qualifier — Data type qualifier flag

off (default) | on

Select this parameter to specify a data type qualifier to apply to the variable.

Parameter Dependencies

If you select this parameter, use parameter **Data type qualifier** to set the data type qualifier to apply to the variable.

Data type qualifier — Data type qualifier for variable

volatile (default) | sting | character vectory

Specify the data type qualifier to apply to the variable in generated code as a string or character vector. Common qualifiers are `volatile`, `const`, `static`, and `register`. The block does not check whether the value that you enter is a valid qualifier.

Data dimension — Number of elements of variable data type

64 (default) | positive integer

Specify the number of elements of the specified data type for the variable as a positive integer.

Initialize memory — Memory initialization flag

off (default) | on

Select this parameter to specify an initial value for the variable.

Parameter Dependencies

If you select this parameter, use parameter **Initial value** to set the initial value.

Initial value — Initial value for variable

0 (default) | scalar | vector | matrix

Specify the initial value for the variable. At run time, the block sets the memory location to this value.

Parameter Dependencies

To enable this parameter, select **Initialize memory**.

Section

Specify the memory section in which to allocate the variable.

Specify memory section — Memory section flag

off (default) | on

Select this parameter to specify a memory section to use for allocating space in memory for the variable.

Parameter Dependencies

If you select this parameter, use parameters **Memory section**, **Bind memory section**, **Section start address** to specify memory section details.

Memory section — Memory section for variable
mySEC1 (default) | string | character vector

Specify the name of the memory section to use for allocating memory for the variable as a string or character vector. Specify a standard memory section or a custom memory section that you declare elsewhere in your code.

Verify that the memory section has enough space to store the variable.

Parameter Dependencies

- To enable this parameter, select **Specify memory section**.
- To bind the specified memory section to a specific start address in memory, select **Bind memory section** and specify the address by entering a value for **Section start address**.

Bind memory section — Bind memory section to start address flag
off (default) | on

Select this parameter to bind a newly created memory section for the variable to a specific start address.

The new memory section specified for **Memory section** is defined when you select this parameter.

Parameter Dependencies

- Select this parameter to enable parameter **Section start address**.
- Do not select this parameter if you are associating the variable with an existing memory section.

Section start address — Start address of memory section for variable
hex2dec('8000') (default) | memory address in decimal or hexadecimal form

Specify the start address to which to bind the memory section for the variable in decimal form or in hexadecimal form with a conversion to decimal as shown by the default value `hex2dec('8000')`. The block does not verify the address. Verify that the address that you specify exists and that it can contain the specified memory section.

Parameter Dependencies

- Enable this parameter by selecting parameter **Bind memory section**.
- Do not specify a value for this parameter if you are associating the variable with an existing memory section.

Version History

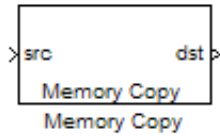
Introduced in R2011a

See Also

Memory Copy

Memory Copy

Copy data from and to memory section



Description

Generated code for the Memory Copy block copies data from and to processor memory as configured by block parameters. When you use this block to copy an individual data element from a source to a destination, the block copies the element from the source, using the source data type, and then casts the data element to the specified destination data type.

Include as many instances of the Memory Copy block in a model as required to manipulate memory on a target processor. Each instance of the block works with one variable, address, or set of addresses provided to the block as input.

Specify the source and destination for a memory copy by using block parameters. You can use block parameters to control other aspects of a memory copy, such as:

- Initialization for memory locations
- Memory stride and offset during run time
- Write operations to memory during program initialization, during program termination, and at every sample time
- Insertion of custom ANSI C source code before and after each memory copy read and write operation (for example, to lock and unlock registers before and after accessing them)

The Memory Copy block performs operations at three periods during program execution:

- Initialization
- Real-time operations
- Termination

You can use block parameters to control when and how the block initializes memory, copies data or variables to and from memory, and terminates copy operations. The parameters enable you to turn on and off memory copy operations in the three periods independently.

Use the Memory Copy block and the Memory Allocate block to manipulate and allocate memory for custom device drivers, such as PCI bus drivers or codec-style drivers.

During simulation, the Memory Copy block does not perform an operation. The block output is not defined.

Ports

Input

src — Input data for copy operation

scalar | vector

The source data for the memory copy operation, specified as a scalar or vector.

Port Dependencies

To use this port as the source for the memory copy operation, set parameter **Copy from** to Input port.

Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

&src — Address of input data for copy operation

scalar | vector

The memory address of source data for the copy operation, specified as a scalar or vector.

Port Dependencies

To use this port as the source for the memory copy operation, set parameter **Copy from** to Specified address and **Specify address source** to Input port. The Copy Memory block converts input port src to &src.

Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

src ofs — Offset for data read during copy operation

scalar | vector

Offset to use for data read during the copy operation, specified as a scalar or vector.

Port Dependencies

To create this port, select parameter **Use offset when reading** and set **Specify offset source** to Input port.

Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

&dst — Address of output data for copy operation

scalar | vector

The memory address to use as the data destination for the copy operation, specified as a scalar or vector.

Port Dependencies

To use this port as the destination for the memory copy operation, set parameter **Copy to** to Specified address and **Specify address source** to Input port. The Copy Memory block converts output port dst to input port &dst.

Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

dst ofs — Offset for data written during copy operation

scalar | vector

Offset to use for data write during the copy operation, specified as a scalar or vector.

Port Dependencies

To create this port, select parameter **Use offset when writing** and set **Specify offset source** to Input port.

Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output

dst — Output data for copy operation
scalar | vector

The data copied, specified as a scalar or vector.

Port Dependencies

To use this port as the destination for the memory copy operation, set parameter **Copy to** to Output port.

Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Parameters

Source

Specify the source sequential memory location for the copy operation. Specify the data type, size, and other attributes of the source variable.

Copy from — Input source for copy operation
Input port (default) | Specified address | Specified source code symbol

Specify the input source for the data read part of the copy operation. Choose from the sources listed in this table.

Source of Data Read	Parameter Value to Specify
src input port	Input port
Memory address	Specified address
Symbol (variable) in source code lookup table	Specified source code symbol

Parameter Dependencies

- If you select Specified address, use **Specify address source** to specify the source of the memory address and **Address** to specify the address.
- If you select Specified source code symbol, use **Source code symbol** to specify the symbol (variable) in the source code symbol table to copy.
- If you select Specified address or Specified source code symbol, change **Data type** to a value other than Inherit from source (the default). If you do not make this change, you receive an error message indicating that the data type cannot be inherited because the input port does not exist.

Specify address source — Source of memory address for input data
Specify via dialog (default) | Input port

Specify the source of the memory address of the input source for the copy operation. To specify a memory address for the source variable, select **Specify via dialog**. That selection enables an **Address** parameter that you use to specify the memory address.

To specify that the block get the address from the input port, select **Input port**. When you select **Input port**, the block input port label changes to `&src`.

Parameter Dependencies

- To enable this parameter, set **Copy from** to **Specified address**.
- If you select **Specify via dialog**, this parameter enables the **Address** parameter, which you use to specify the address of the source variable.
- If you select **Specify via dialog**, set **Data type** to a value other than **Inherit from source** (the default). If you do not make this change, you receive an error message indicating that the data type cannot be inherited because the input port does not exist.
- If you select **Input port**, specify a data type for the **Data type** parameter.

Address — Memory address of source data

`hex2dec('00001000')` (default) | memory address in decimal or hexadecimal form with a conversion to decimal

Specify the memory address of the source data in decimal form or in hexadecimal form with a conversion to decimal as shown by the default value `hex2dec('00001000')`.

This example converts `0x1000` to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you can specify the address as `4096` or `hex2dec('1000')`.

Parameter Dependencies

To enable this parameter, set **Copy from** to **Specified address** and **Specify address source** to **Specify via dialog**.

Source code symbol — Symbol in source code symbol table

`myVariableSrc` (default) | string | character vector

Specify the symbol (variable) in the source code symbol table to copy. The symbol that you specify must exist in the symbol table for your program. The block does not verify whether the symbol exists in the symbol table and whether you specify the symbol with valid syntax. Enter text that specifies the symbol exactly as it appears in your code.

Parameter Dependencies

- To enable this parameter, set **Copy from** to **Specified source code symbol**.
- Set **Data type** to a value other than **Inherit from source** (the default). If you do not make this change, you receive an error message indicating that the data type cannot be inherited because the input port does not exist.

Data type — Data type of data being copied

`uint8` (default) | `double` | `single` | `int8` | `int16` | `uint16` | `int32` | `uint32` | `int64` | `uint64` | `boolean` | **Inherit from input port**

Specify the data type of the source data being copied. To inherit the data type from the `src` input port, select **Inherit from input port**.

Data length — Number of data elements to copy

1 (default) | positive integer

Specify the number of elements to copy from the source location. Each element has the data type specified by the **Data type** parameter.

Use offset when reading — Use offset when reading input

off (default) | on

Specify whether the block uses an offset when reading input. The offset value is in elements of the specified data type. Specify the source of the offset by using the **Specify offset source** parameter.

Parameter Dependencies

- If you select this parameter, use **Specify offset source** to specify the source of the offset.
- Use **Offset** to specify the offset value.

Specify offset source — Source of input offset

Specify via dialog (default) | Input port

Specify the source of the input offset for the copy operation. To specify an offset value, select **Specify via dialog**. That selection enables an **Offset** parameter that you use to specify the offset value.

To specify for the block to get the offset from an input port, select **Input port**. When you select **Input port**, the block creates an input port labeled `src ofs` and reads the offset value from that port. The `src ofs` port enables your program to change the offset dynamically during program execution.

Parameter Dependencies

To enable this parameter, select **Use offset when reading**.

Offset — Number of values to skip before copying first value to destination

0 (default) | positive integer

Before copying the first value to the destination, specify the number of values to skip.

Parameter Dependencies

To enable this parameter, select **Use offset when reading** and set **Specify offset source** to **Specify via dialog**.

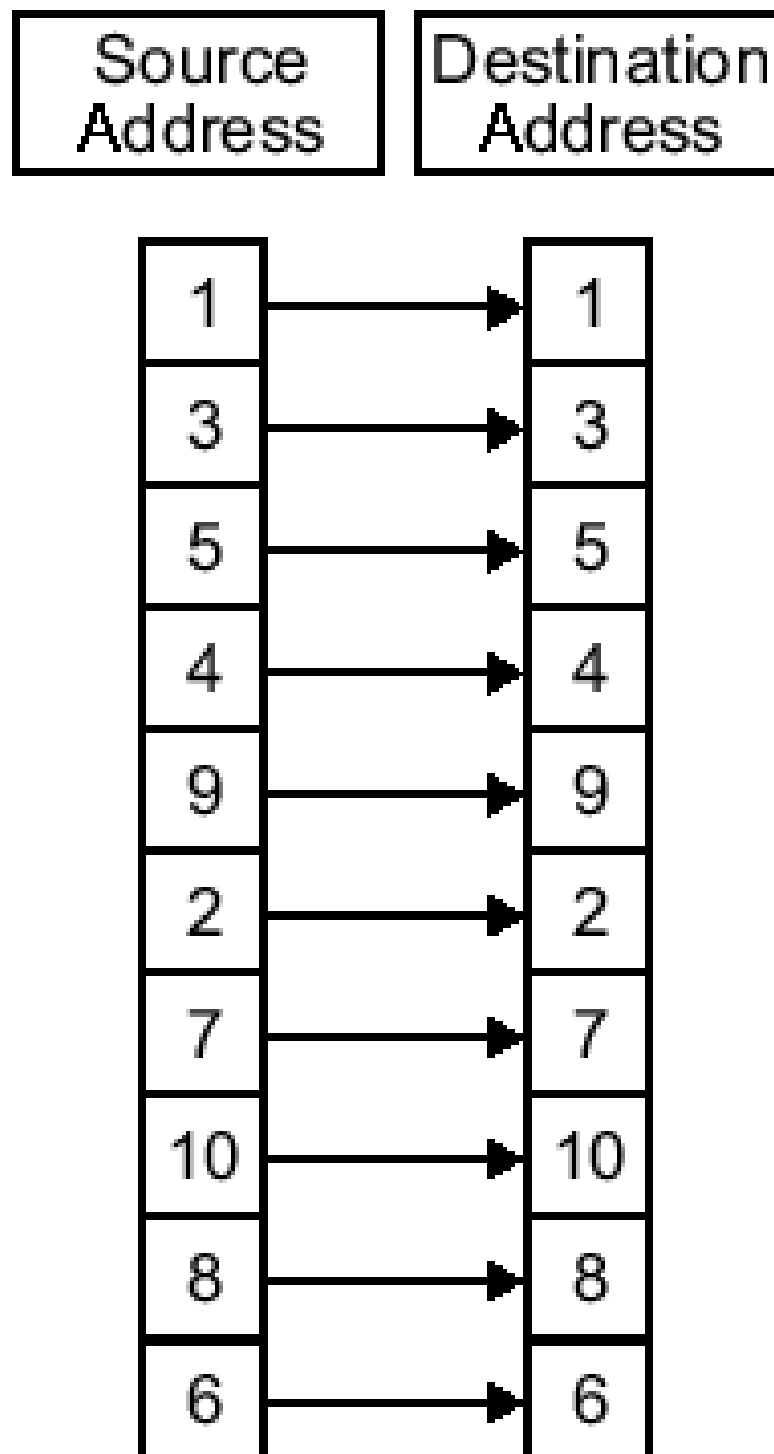
Stride — Spacing for reading input

1 (default) | positive integer

Specify the spacing for reading the input. By default, the stride value is one, meaning that the generated code reads the input data sequentially. When you add a stride value that is not equal to one, when reading input data, the generated code skips spaces in the source address equal to the stride.

These figures show the stride concept. In the first figure, data is copied without a stride. The second figure shows the results of a stride value of two. You can specify a stride value for the block output

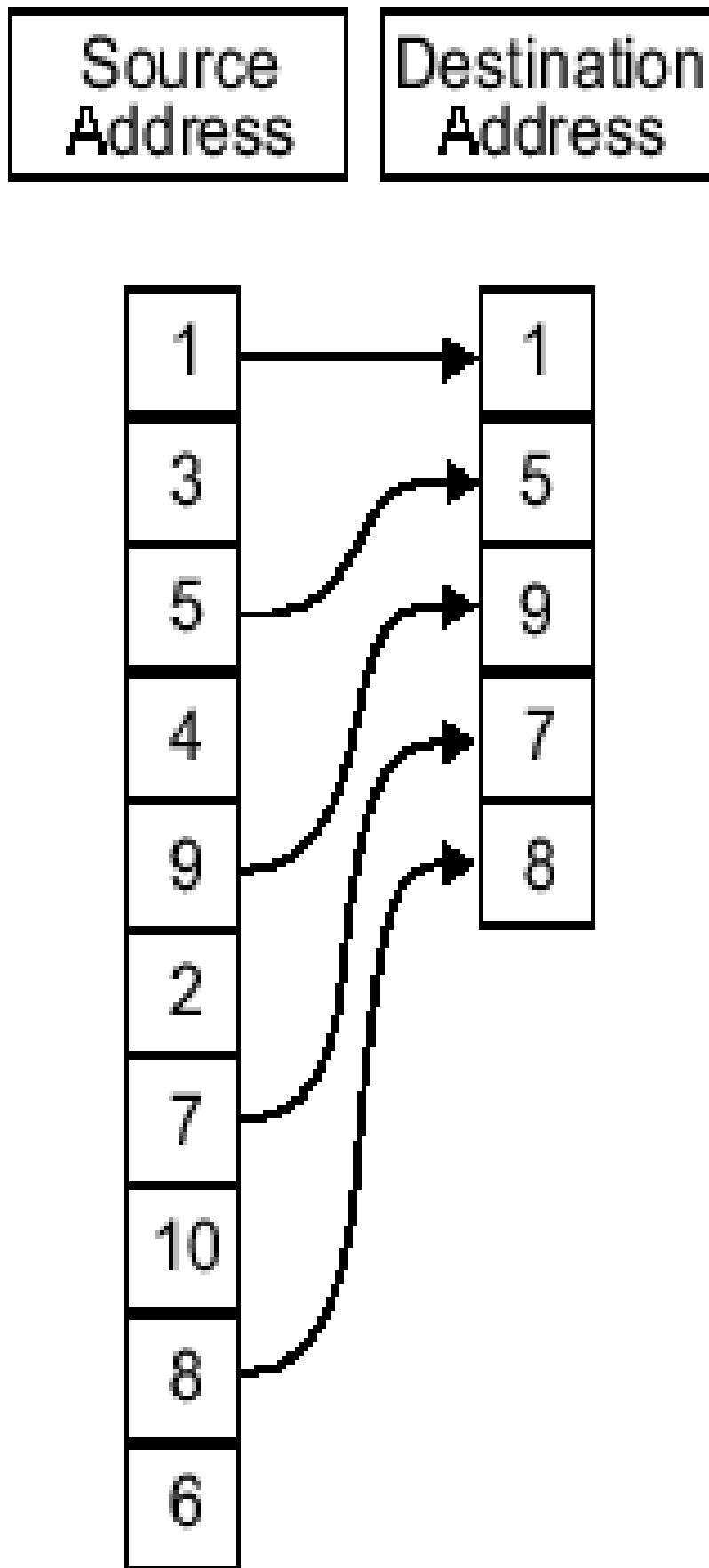
with parameter **Stride** on the **Destination** tab. You can also compare stride with the offset to see the differences.



Input Stride = 1

Output Stride = 1

Number of Elements Copied = 10



Destination

Specify the destination memory location for the copy operation. Specify the attributes of the destination.

Copy to — Type of output destination for copy operation

Output port (default) | Specified address | Specified destination code symbol

Specify the type of output destination for the copy operation. Select one of the values listed in this table.

Parameter Value	Destination of Data Write
Output port	Block dst output port
Specified address	Memory location specified by parameters Specify address destination and Address
Specified source code symbol	Symbol (variable) specified by parameter Source code symbol

Parameter Dependencies

- If you select **Specified address**, use **Specify address destination** to specify the destination memory location.
- If you select **Specified source code symbol**, use **Destination code symbol** to specify the symbol (variable) in the source code symbol table to which to copy the variable.

Specify address source — Source of memory address for output destination

Specify via dialog (default) | Input port |

Specify the source of the destination memory address of the variable for the copy operation. To specify a memory address for the variable, select **Specify via dialog**. That selection enables an **Address** parameter that you use to specify the memory address. To specify that the block get the address from an input port, select **Input port**. When you select **Input port**, the block creates an input port labeled &dst. Changing the address dynamically means that you can use the block to copy different variables by providing the variable address from an upstream block in the model.

Parameter Dependencies

- To enable this parameter, set **Copy to** to **Specified address**.
- If you select **Specify via dialog**, this parameter enables the **Address** parameter, which you use to specify the address of the destination variable.

Address — Memory address of destination variable

hex2dec('00002000') (default) | memory address in decimal or hexadecimal form with a conversion to decimal

Specify the memory address of the destination variable in decimal form or in hexadecimal form with a conversion to decimal as shown by the default value `hex2dec('00001000')`.

This example converts 0x2000 to decimal form.

```
8192 = hex2dec('2000');
```

For this example, you can specify the address as 8192 or `hex2dec('2000')`.

Parameter Dependencies

To enable this parameter, set **Copy to** to Specified address and **Specify address source** to Specify via dialog.

Source code symbol — Symbol in source code symbol table

myVariableDst (default) | string | character vector

Specify the symbol (variable) in the source code symbol table to which to copy the variable. The symbol that you specify, must exist in the symbol table for your program. The block does not verify whether the symbol exists in the symbol table and whether you specify the symbol with valid syntax. Enter text that specifies the symbol exactly as it appears in your code.

Parameter Dependencies

To enable this parameter, set **Copy to** to Specified source code symbol.

Data type — Data type of variable

uint32 (default) | double | single | int8 | uint8 | int16 | uint16 | uint32 | int64 | uint64 | boolean | Inherit from source

Specify the data type of the source variable. To inherit the data type from the source variable, select **Inherit from source**.

Use offset when writing — Use offset when writing output

off (default) | on

Specify whether the block uses an offset when writing output. The offset value is in elements of the specified data type. Specify the source of the offset by using the **Specify offset source** parameter.

Parameter Dependencies

If you select this parameter, use **Specify offset source** to specify the source of the offset. Use **Offset** to specify the offset value.

Specify offset source — Source of offset for output destination

Specify constant value (default) | Specify source code symbol

Specify the source of the offset for the output destination for the copy operation. To specify an offset value for the destination variable, select **Specify via dialog**. That selection enables an **Offset** parameter that you use to specify the offset value.

To specify that the block get the offset from the input port, select **Input port**. When you select **Input port**, the block adds an input port labeled **dst ofs** and reads the offset value from that port. The **dst ofs** port enables your program to change the offset dynamically during execution.

Parameter Dependencies

To enable this parameter, select **Specify offset source**.

Offset — Number of values to skip before writing first value to destination

0 (default) | positive integer

Before writing the first value to the destination, specify the number of values to skip.

Parameter Dependencies

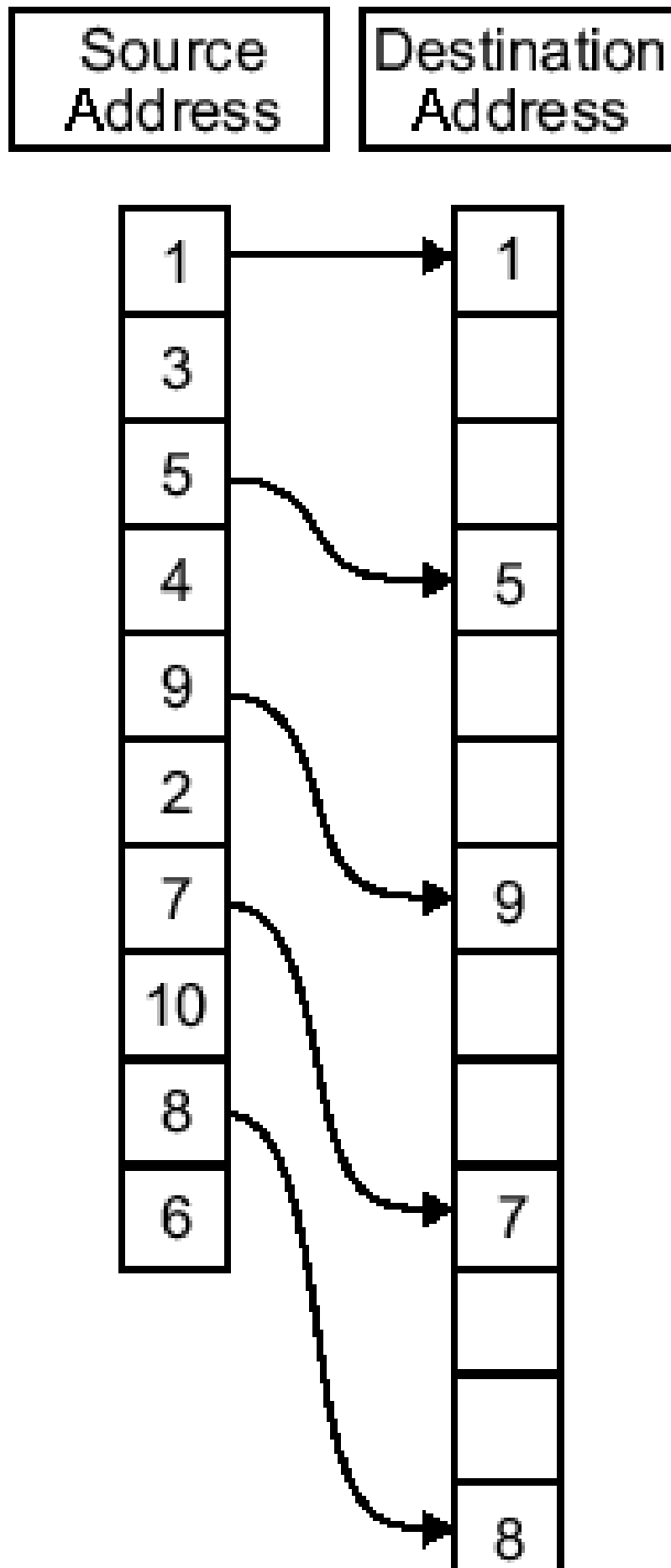
To enable this parameter, select **Use offset when writing** and set **Specify offset source** to **Specify via dialog**.

Stride — Spacing for writing output

1 (default) | positive integer

Specify the spacing for writing the output. By default, the stride value is one meaning that the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value that is not equal to one, when writing input data, the generated code skips spaces in the destination address equal to the stride.

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 an input stride and an output stride at the same time to enable manipulating memory more fully.



Sample time — Rate of memory copy

inf (default) | scalar

Specify the rate at which the memory copy operation occurs in seconds. To use a constant sample time, specify Inf. To inherit the sample time from the input port or, when the block does not have an input port, from the Simulink model, specify -1.

Options

Configure parameters that control the copy process.

Set memory value at initialization — Set memory address during initialization

off (default) | on

Specify whether to initialize the memory address to a specific value during program initialization.

Parameter Dependencies

If you select this parameter, use a combination of parameters to configure the initialization value.

What to Configure	Parameter
Source of the initialization value	Specify initialization value source
Initialization value as a constant	Initialization value (constant)
Initialization value as a variable	Initialization value (source code symbol)
Initialization value as a mask to manipulate register contents at the bit level	Apply initialization value as mask
Apply a mask value	Bitwise operator

Specify initialization value source — Source of initialization value

Specify constant value (default) | Specify source code symbol

Specify the source of the initial value. To configure the source for initializing memory as a specific value, select **Specify constant value**. To configure the source as a variable (a symbol), select **Specify source code symbol**.

Parameter Dependencies

- To enable this parameter, select **Set memory value at initialization**.
- Use **Initialization value (constant)** or **Initialization value (source code symbol)** to specify the initial value.

Initialization value (constant) — Constant initialization value

1 (default) | scalar

Specify a constant value.

Parameter Dependencies

To enable this parameter, select **Set memory value at initialization** and set **Set initialization value source** to **Specify constant value**.

Initialization value (source code symbol) — Symbol in source code symbol table

myInitValueVariable (default) | string | character vector

Specify the symbol (variable) in the source code symbol table to use for the initialization value. The symbol that you specify must exist in the symbol table for your program. The block does not verify whether the symbol exists in the symbol table and whether you specify the symbol with valid syntax. Enter text that specifies the symbol exactly as it appears in your code.

Parameter Dependencies

To enable this parameter, select **Set memory value at initialization** and set **Set initialization value source** to `Specify source code symbol`.

Apply initialization value as mask — Apply initialization value as mask
off (default) | on

Specify whether to use the initialization value as a mask to manipulate register content at the bit level. Your initialization value is treated as a string of bits for the mask.

To define how to apply the mask value, specify a value for the **Bitwise operator** parameter.

To use your initialization value as a mask, the output from the copy must be a specific address. The output:

- Can be a symbol
- Cannot be an output port

Parameter Dependencies

If you select this parameter, use **Bitwise operator** to define how to apply the mask value.

Bitwise operator — Type of bitwise operation
bitwise AND (default) | bitwise OR | bitwise exclusive OR | left shift | right shift

Specify the type of bitwise operation to use as a mask to manipulate the memory value. Applying a mask to the copy process means that you can select individual bits in the result. For example, by applying a mask, you can read the value of the fifth bit.

Select one of the bitwise operations in this table.

Bitwise Operation	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.

Parameter Dependencies

To enable this parameter, select **Apply initialization value as mask**.

Set memory value at termination — Copy memory during program termination

off (default) | on

Specify that your program copy memory during program termination. Copying a value in memory during termination occurs in addition to a copy during program initialization.

Parameter Dependencies

If you select this parameter, you can use **Set memory value only at initialization/termination** to limit copy operations to occur during program initialization and termination only.

Termination value — Termination value

1 (default) | scalar | vector | matrix

Specify a value to write to memory during program termination.

Parameter Dependencies

To enable this parameter, select **Set memory value at termination**.

Set memory value only at initialization/termination — Copy memory value during program initialization and termination only

off (default) | on

Specify whether to perform copies during program initialization and termination only. When this parameter is cleared, the block performs copies during initialization, real-time operations, and termination. If you select this parameter, the block performs copies during initialization and termination only.

Insert custom code before memory write — Insert custom code before memory write

off (default) | on

Specify whether the code generator inserts custom ANSI C code immediately before the program writes to the specified memory location. You can use this parameter and **Insert custom code after memory write** to lock and unlock registers before and after accessing them. For example, some processors have registers that you might need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

Parameter Dependencies

If you select this parameter, use **Custom code** to specify the custom ANSI C code to insert into the generated code immediately before the memory write operation.

Insert custom code after memory write — Custom code after memory write flag

off (default) | on

Specify whether the code generator inserts custom ANSI C code immediately after the program writes to the specified memory location. You can the **Insert custom code before memory write** and this parameter to lock and unlock registers before and after accessing them. For example, some processors have registers that you might need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

Parameter Dependencies

If you select this parameter, use **Custom code** to specify the custom ANSI C code to insert into the generated code immediately after the memory write operation.

Custom code —

`/* Custom Code Before Write*/ or /* Custom Code After Write*/ (default) | string | character vector`

Specify custom ANSI C code to insert into the generated code immediately before or immediately after the memory write operation. Code that you specify appears in the generated code exactly as you enter it.

Parameter Dependencies

To enable this parameter, select **Insert custom code before memory write** or **Insert custom code after memory write**.

Version History

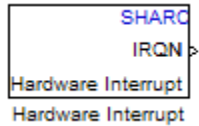
Introduced in R2011a

See Also

Memory Allocate

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: preemptable-1, non-preemptable-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. 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 preemptable - 1, non-preemptable - 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 preemptable and the priority 39 interrupt can be preempted.

Enable simulation input

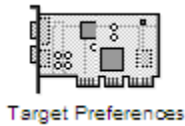
When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Version History

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:

- “Configure Target Hardware Resources”
- “Hardware Implementation Pane”

Version History

Introduced in R2013a

UDP Receive

Receive UDP packet



Libraries:

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. Due to the nature of the UDP transmission protocol, the receiver is not guaranteed to receive all the data packets that you send using the UDP Send block. 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 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 the extra `.dll` files. The `packNGo` function creates a ZIP file that contains the pieces required to run or rebuild this code. For more details, see “How To Run a Generated Executable Outside MATLAB” (DSP System Toolbox).

Ports

Output

Port_1 — UDP packet
vector

UDP packet, specified as a data vector, which is received from an IP network port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Local IP port — Number of IP port
25000 (default) | scalar in the range [1, 65535]

Specify the IP port number on which to receive UDP packets. This parameter is tunable in the generated code but is not tunable during simulation.

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) — IP address from which to accept packets
'0.0.0.0' (default) | IP address

Specify the IP address from which to accept UDP packets. Specify a specific IP address to block UDP packets from other addresses. To accept packets from any IP address, specify '0.0.0.0'.

Receive buffer size (bytes) — Size of buffer that receives UDP packets

8192 (default) | scalar in the range [1, 67108864]

Specify the size of the buffer, in bytes, that receives the UDP packets. Make the buffer large enough to avoid data loss caused by buffer overflows.

Maximum length for Message — Maximum length of output data

255 (default) | scalar

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.

Parameter Dependencies

If you disable **Output variable-size signal**, the block output is the length specified by this parameter.

Data type for Message — Data type of message

uint8 (default) | single | double | int8 | int16 | int32 | int64 | uint16 | uint32 | uint64 | boolean | fixed point | enumerated | bus

Specify the data type of the vector elements in the message output. Match the data type to the data input used to create the UDP packets.

Message is complex — Message data complexity

off (default) | on

Specify whether the block receives a message as complex data. Select this parameter to receive a message as complex data. Clear this parameter if a received message is real data.

Output variable-size signal — Message output that varies in length

on (default) | off

Specify whether your model supports signals of varying length. If your model supports signals of varying length, select this parameter. 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, clear this parameter. In that case:

- The block emits a fixed-length output that is the same length as specified by **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 **Data type for Length** parameter is enabled.

Parameter Dependencies

If you disable this parameter, the block emits output that is the length specified by **Maximum length for Message**.

Blocking time (seconds) — Number of seconds to wait for UDP packet
`inf` (default) | scalar

For each sample, specify the number of seconds to wait for a UDP packet before returning control to the scheduler. To wait indefinitely, specify `inf`.

Note This parameter applies to the Embedded Coder UDP Receive block only.

Sample time (seconds) — Frequency of calls to block
`0.01` (default) | scalar

Specify the frequency at which the scheduler calls the UDP Receive block, in seconds. Enter a value greater than zero. In real-time operation, setting this parameter to a smaller value reduces the likelihood of dropped UDP messages.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Byte Pack | Byte Reversal | Byte Unpack | UDP Send

UDP Send

Send UDP packet



Libraries:

Embedded Coder / Embedded Targets / Host Communication

Description

The UDP Send block sends data packets over the network using the User Datagram Protocol (UDP). UDP is a simple message-based connectionless protocol and does not have an end-to-end connection. The protocol sends data packets in one direction from source to destination without verifying the readiness of the receiver. The protocol has no handshaking mechanism. The data packets can get dropped for several reasons. There is no acknowledgement, retransmission, or timeout in UDP. However, UDP is a very simple transmission protocol and is suitable for time-sensitive applications where dropping packets is preferable to waiting for packets delayed due to retransmission.

The UDP Send block transmits an input data vector as a UDP packet to a remote IP network port. The remote IP port number to which the block sends the UDP packets is tunable in the generated code.

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 of 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).

Ports

Input

Port_1 — UDP packet

scalar (default) | vector

UDP packet, specified as a data vector, which is transmitted to an IP network port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Remote IP address ('255.255.255.255' for broadcast) — IP address from which to accept UDP packets

'255.255.255.255' (default) | IP address | string

Specify the IP address or hostname to which to send UDP packets. If you specify a hostname, specify it as a string. To broadcast a UDP packet, specify '255.255.255.255'.

Remote IP port — Number of remote IP port
25000 (default) | scalar in the range [1, 65535]

Specify the IP port number to which to send UDP packets. This parameter is tunable in the generated code but is not tunable during simulation.

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 — Source of local IP port
Automatically determine (default) | Specify via dialog

Specify whether the block uses a local port number that the system assigns or that you specify by using the **Local IP port** parameter. To let the system assign the port number, select **Automatically determine**. If the receiving address expects UDP packets from a specific port number, select **Specify via dialog** and specify the port number by using the **Local IP port** parameter.

Parameter Dependencies

To enable the **Local IP port** parameter, select **Specify via dialog**.

Local IP port — Number of local IP port
-1 (default) | scalar in the range [1, 65535]

Specify the IP port number from which the block sends UDP packets. Use this parameter when the receiving address expects messages from a specific port number.

Parameter Dependencies

To enable this parameter, set **Local IP port source** to **Specify via dialog**.

Send buffer size (bytes) — Size of buffer that sends UDP packets
8192 (default) | scalar in the range [1, 67108864]

Specify the size of the buffer, in bytes, that sends the UDP packets. Make the buffer large enough to avoid data loss caused by buffer overflows.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

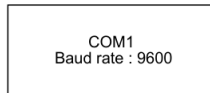
Generate C and C++ code using Simulink® Coder™.

See Also

Byte Pack | Byte Reversal | Byte Unpack | UDP Receive

Serial Configuration

Configure parameters for serial port



Libraries:

Instrument Control Toolbox

Motor Control Blockset / Protection and Diagnostics

C2000 Microcontroller Blockset / Host Communication

Description

The Serial Configuration block configures parameters for a serial port that you can use to send and receive data. Use this block to set the parameters of your serial port before you set up the Serial Receive and the Serial Send blocks.

Note You must configure your serial port parameters using the Serial Configuration block before you specify the Serial Receive and Serial Send block parameters.

Other Supported Features

- The Serial Configuration block supports the use of Simulink Accelerator mode, but not Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The Serial Configuration block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The Serial Configuration block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder.

For more information on these features, see the “Simulink” documentation.

Parameters

Port — Serial communication port
available communication ports

Serial port on your machine that you want to configure. Use this configured port to send and receive data with your Serial Send and Serial Receive blocks. If you have not configured a port, the block returns an error when you run your model.

Note Each Serial Send and Serial Receive block must have a configured Serial Configuration block. If you use multiple serial ports in your simulation, you must configure each port using a separate Serial Configuration block.

Programmatic Use

Block Parameter: Port

Type: character vector, string

Baud rate — Communication speed
9600 (default) | positive integer

Rate at which bits are transmitted for the serial interface, in bits per second.

Programmatic Use
Block Parameter: BaudRate
Type: character vector, string
Values: positive integer
Default: '9600'

Data bits — Number of bits to represent one character of data
8 (default) | 5 | 6 | 7

Number of data bits to transmit over the serial interface.

Programmatic Use
Block Parameter: DataBits
Type: character vector, string
Values: '5' | '6' | '7' | '8'
Default: '8'

Parity — Parity bit type
none (default) | even | odd

Parity bit type added to data transmitted by serial port. You can use this parameter to add a parity bit (also referred to as a check bit) to your data. Adding a parity bit to a string of binary code is a method of detecting errors in data transmission by ensuring that the total number of 1-bits is even or odd.

The value of the parity bit is determined by the number of 1s in a given set of bits and is set as follows.

Parity Bit Type	Parity Bit Value	
	If number of 1s is even	If number of 1s is odd
none	No parity bit set	No parity bit set
even	0	1
odd	1	0

Note Starting in R2021a, the **Parity** parameter no longer supports **mark** or **space**. For more information, see “Parity parameter no longer supports mark or space in Serial Configuration block” on page 3-91.

Programmatic Use
Block Parameter: Parity
Type: character vector, string
Values: 'none' | 'even' | 'odd'
Default: 'none'

Stop bits — Pattern of bits that indicates the end of a character
1 (default) | positive scalar

Number of bits used to indicate the end of a byte. This parameter depends on the value you select for the **Data bits** parameter. If you select data bits 6, 7, or 8, the default value is 1 and the other

available choice is 2. If you select data bit 5, the default value is 1 and the other available choice is 1.5.

Programmatic Use

Block Parameter: StopBits

Type: character vector, string

Values: positive scalar

Default: '1'

Byte order — Sequential order of bytes

little-endian (default) | big-endian

Sequential order in which bytes are arranged into larger numerical values. If the byte order is little-endian, then the instrument stores the first byte in the first memory address. If the byte order is big-endian, then the instrument stores the last byte in the first memory address.

Configure the byte order to the appropriate value for your instrument before performing a read or write operation. Refer to your instrument documentation for information about the order in which it stores bytes.

Programmatic Use

Block Parameter: ByteOrder

Type: character vector, string

Values: 'little-endian' | 'big-endian'

Default: 'little-endian'

Flow control — Mode for managing data transmission rate

none (default) | hardware

Process of managing the rate of data transmission on your serial port. Select none to have no flow control or hardware to let your hardware determine the flow control.

Programmatic Use

Block Parameter: FlowControl

Type: character vector, string

Values: 'none' | 'hardware'

Default: none

Timeout — Allowed time to complete operations

10 (default) | positive scalar

Amount of time that the model waits for data during each simulation time step.

Programmatic Use

Block Parameter: Timeout

Type: character vector, string

Values: positive scalar

Default: '10'

Version History

Introduced in R2008a

R2021a: Parity parameter no longer supports mark or space in Serial Configuration block

Errors starting in R2021a

The **mark** and **space** options for the **Parity** parameter are no longer supported in the Serial Configuration block. Valid values for **Parity** are **none** (default), **even**, and **odd**.

If you try to open an existing model that has the **Parity** value set to **mark** or **space**, MATLAB returns a warning and changes the parameter to the default value **none**.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux).

Serial Receive

Receive binary data over serial port



Libraries:

Instrument Control Toolbox

Motor Control Blockset / Protection and Diagnostics

C2000 Microcontroller Blockset / Host Communication

Description

The Serial Receive block configures and opens an interface to the specified serial port. The configuration and initialization occur once at the start of the model's execution. The block acquires data from the serial port during the model's run time. You can use only one Serial Receive block at a time to receive data from a specific serial port.

Note You must configure your serial port parameters using the Serial Configuration block before you specify the Serial Receive block parameters.

This block has no input ports. It has one or two output ports based on whether you select blocking or non-blocking mode. If you select blocking mode, the block has one output port, **Data**, corresponding to the data it receives. If you do not select blocking mode, the block has two output ports, **Data** and **Status**.

This block uses a First In, First Out (FIFO) buffer to receive data from the serial port. At each time step, the **Data** port returns the requested values from the buffer. In non-blocking mode, the **Status** port indicates if the block has received new data. If the **Status** port displays 1, new data is available and if the **Status** port displays 0, no new data is available.

Other Supported Features

- The Serial Receive block supports the use of Simulink Accelerator mode, but not Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The Serial Receive block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The Serial Receive block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder.

For more information on these features, see the “Simulink” documentation.

Ports

Output

Data — Data received

vector | matrix | array

Data received by the block from the serial port, returned as a vector, matrix, or array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Status — New data available

`false` or `0` (default) | `true` or `1`

New data available status, returned as numeric or logical `1` (`true`) or `0` (`false`). If this port returns `1`, new data is available to be read.

Dependencies

To enable this port, clear the parameter.

Data Types: `Boolean`

Parameters

Port — Serial communication port

available communication ports

Serial port on your machine that you want to receive data from. Select a port from the available ports and then configure the port using the Serial Configuration block. If you have not configured a port, the block returns an error when you run your model.

Note Each Serial Receive block must have a configured Serial Configuration block. If you use multiple serial ports in your simulation, you must configure each port using a separate Serial Configuration block.

Programmatic Use

Block Parameter: Port

Type: character vector, string

Data type — Output data type

`uint8` (default) | `single` | `double` | `int8` | `int16` | `uint16` | `int32` | `uint32`

Data type that the block receives from the serial port, specified as a MATLAB numeric data type.

Programmatic Use

Block Parameter: DataType

Type: character vector, string

Values: `'uint8'` | `'single'` | `'double'` | `'int8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'`

Default: `'uint8'`

Header — Header

numeric array | integer from 0 to 255, inclusive

If this parameter is selected, you can specify the header that indicates the beginning of your data block. The simulation disregards data that occurs before the header. The header data is not sent to the output port. By default, this parameter is not selected and no header is specified.

The numeric array specified in this parameter is the `uint8` integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	"#"	uint8('#')	[35]
Numeric	"81"	uint8('81')	[56 49]
Alphabet	"Start"	uint8('Start')	[83 116 97 114 116]

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use

Block Parameter: ToggleHeader

Type: character vector, string

Values: 'on' | 'off'

Default: 'off'

Block Parameter: Header

Type: character vector, string

Values: integer array

End token — Terminator

<none> (default) | CR ('\r') | LF ('\n') | CR/LF ('\r\n') | NULL ('\0') | Custom Terminator

If this parameter is selected, you can specify the end token or terminator that indicates the end of your data block. The simulation considers any data that occurs after the end token as a new data block. The end token data is not sent to the output port. This end token must match the one in the data you are reading from your serial port, if it has one.

If you select Custom Terminator, you can specify your own end token value.

Programmatic Use

Block Parameter: ToggleTerminator

Type: character vector, string

Values: 'on' | 'off'

Default: 'off'

Block Parameter: Terminator

Type: character vector, string

Values: '<none>' | 'CR ('\r')' | 'LF ('\n')' | 'CR/LF ('\r\n')' | 'NULL ('\0')' | 'Custom Terminator'

Default: '<none>'

Custom terminator — Custom terminator

numeric array | integer from 0 to 255, inclusive

Custom terminator that indicates the end of your data block, used when **End token** is set to 'Custom Terminator'. The simulation considers any data that occurs after the terminator as a new data block. The terminator data is not sent to the output port.

The numeric array specified in this parameter is the uint8 integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	"#"	uint8('#')	[35]
Numeric	"81"	uint8('81')	[56 49]
Alphabet	"End"	uint8('End')	[69 110 100]

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use

Block Parameter: CustomTerminator

Type: character vector, string

Values: integer array

Input Format — Format of data read

Column major (default) | Row major

Format of the data that the block receives from the serial port, specified as Row major or Column major.

Programmatic Use

Block Parameter: InputFormat

Type: character vector, string

Values: 'Row major' | 'Column major'

Default: 'Column major'

Data size — Number of values read

[1 1] (default) | numeric array

Output data size, or the number of values that should be read at each simulation time step. This parameter is specified as a multidimensional numeric array. The data size does not include the header or terminator values.

Programmatic Use

Block Parameter: DataSize

Type: character vector, string

Values: integer array

Default: '[1 1]'

Enable blocking mode — Simulation waits while receiving data

on (default) | off

This parameter has the simulation wait while the block receives data. When new data becomes available, the simulation continues from the next time step. Clear the check box if you do not want the read operation to cause the simulation to wait.

If you enable blocking mode, the simulation waits for the requested data to become available. The model waits for up to the amount of time specified by the **Timeout** parameter in the Serial Configuration block. If new data does not become available during a simulation, you can return an error by selecting the Error option for the **Action when data is not available** parameter.

If you do not enable blocking mode, the simulation runs continuously and the block has two output ports, **Status** and **Data**. The **Data** port contains the requested set of data at each time step. The **Status** port contains 0 or 1 based on whether it received new data at the given time step.

Programmatic Use**Block Parameter:** EnableBlockingMode**Type:** character vector, string**Values:** 'on' | 'off'**Default:** 'on'**Action when data is not available** — Action to take when data not available

Output last received value (default) | Output custom value | Error

Action the block should take when data is not available. Available options are:

- **Output last received value** — Block returns the value it received at the preceding time step when it does not receive data at current time step.
- **Output custom value** — Block returns any user-defined value when it does not receive current data. Define the custom value in the **Custom value** field.
- **Error** — Block returns an error when it does not receive current data. You must select **Enable blocking mode** to use this option.

Programmatic Use**Block Parameter:** ActionDataUnavailable**Type:** character vector, string**Values:** 'Output last received value' | 'Output custom value' | 'Error'**Default:** 'Output last received value'**Custom value** — Custom output value when data unavailable

0 (default) | numeric

Custom value for the block to output when it does not receive new data. The custom value can be a scalar or value equal to the size of data that it receives (specified by **Data size** parameter). You must select **Output custom value** as the **Action when data is unavailable** to set this parameter.

Programmatic Use**Block Parameter:** CustomValue**Type:** character vector, string**Values:** numeric**Default:** '0'**Block sample time** — Sampling time

0.01 (default) | positive numeric

Sampling time of the block during the simulation. This is the rate at which the block is executed during simulation.

Programmatic Use**Block Parameter:** SampleTime**Type:** character vector, string**Values:** positive numeric**Default:** '0.01'

Version History

Introduced in R2008a

Extended Capabilities

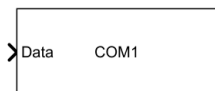
C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux).

Serial Send

Send binary data over serial port



Libraries:

Instrument Control Toolbox

Motor Control Blockset / Protection and Diagnostics

C2000 Microcontroller Blockset / Host Communication

Description

The Serial Send block configures and opens an interface to the specified serial port. The configuration and initialization occur once at the start of the model's execution. The block sends data from the model to the serial port during the model's run time. You can use multiple Serial Send blocks at a time to send data to a specific serial port.

Note You must configure your serial port parameters using the Serial Configuration block before you specify the Serial Send block parameters.

The Serial Send block has one input port that accepts both 1-D vector and matrix data. This block has no output ports. The block inherits the data type from the signal at the input port. Valid data types are `single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`.

Other Supported Features

- The Serial Send block supports the use of Simulink Accelerator mode, but not Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The Serial Send block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The Serial Send block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder.

For more information on these features, see the “Simulink” documentation.

Ports

Input

Data — Data values to send

vector | matrix | array

Data values to send from the block over your serial port, specified as a vector, matrix, or array. Set the parameters for this block before you send data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Parameters

Port — Serial communication port
available communication ports

Serial ports on your machine that you want to send data to. Select a port from the available ports and then configure the port using the Serial Configuration block. If you have not configured a port, the block returns an error when you run your model.

Note Each Serial Send block must have a configured Serial Configuration block. If you use multiple serial ports in your simulation, you must configure each port using a separate Serial Configuration block.

Programmatic Use
Block Parameter: Port
Type: character vector, string

Header — Header
numeric array | integer from 0 to 255, inclusive

Header that indicates the beginning of your data block. The Serial Send block adds the header in front of the data before sending it over the serial port. By default, no header is specified.

The numeric array specified in this parameter is the `uint8` integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	"#"	<code>uint8('#')</code>	[35]
Numeric	"81"	<code>uint8('81')</code>	[56 49]
Alphabet	"Start"	<code>uint8('Start')</code>	[83 116 97 114 116]

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use
Block Parameter: Header
Type: character vector, string
Values: integer array

Terminator — Terminator
<none> (default) | CR ('\\r') | LF ('\\n') | CR/LF ('\\r\\n') | NULL ('\\0') | Custom Terminator

Terminator that indicates the end of your data block. The Serial Send block appends the terminator to the data before sending it over the serial port.

If you select Custom Terminator, you can specify your own terminator value.

Programmatic Use**Block Parameter:** Terminator**Type:** character vector, string**Values:** '<none>' | 'CR ('\r')' | 'LF ('\n')' | 'CR/LF ('\r\n')' | 'NULL ('\0')' | 'Custom Terminator'**Default:** '<none>'**Custom terminator** — Custom terminator

numeric array | integer from 0 to 255, inclusive

Custom terminator that indicates the end of your data block. The Serial Send block appends the terminator to the data before sending it over the serial port.

The numeric array specified in this parameter is the `uint8` integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	"#"	<code>uint8('#')</code>	[35]
Numeric	"81"	<code>uint8('81')</code>	[56 49]
Alphabet	"End"	<code>uint8('End')</code>	[69 110 100]

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use**Block Parameter:** CustomTerminator**Type:** character vector, string**Values:** integer array**Enable blocking mode** — Simulation waits while sending data

on (default) | off

This parameter has the simulation wait while the block sends data. Clear the check box if you do not want the write operation to cause the simulation to wait.

If you enable blocking mode, the simulation waits for the data to be sent. If you do not enable blocking mode, the simulation runs continuously.

Programmatic Use**Block Parameter:** EnableBlockingMode**Type:** character vector, string**Values:** 'on' | 'off'**Default:** 'on'

Version History

Introduced in R2008a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux).

Interprocess Data Read

Receive messages from another processor using interprocess communication channel



Libraries:

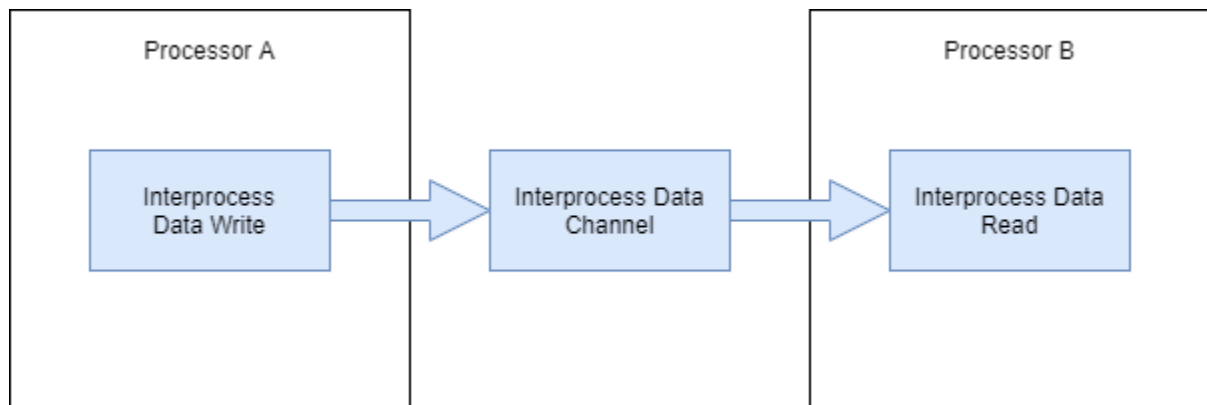
SoC Blockset / Processor Interconnect
 C2000 Microcontroller Blockset / Target Communication
 Embedded Coder Support Package for Infineon AURIX TC3x
 Microcontrollers / Target Communication
 Embedded Coder Support Package for Infineon AURIX TC4x
 Microcontrollers / Target Communication

Description

The Interprocess data Read block asynchronously receives messages from another processor in an SoC using an interprocess data channel. The Interprocess Data Read block connects to an Interprocess Data Channel block that similarly connects to an Interprocess Data Write block contained in a separate processor reference model. In simulation, data from another processor is asynchronously received and processed in the processor containing the Interprocess Data Read block and associated asynchronous task. This diagram shows a generalized view of the interprocessor data channel connection.

Note

- Ensure that the number of buffers specified must same at both the Tx and Rx side.
- If you are using multiprocessor modeling approach with single model triggering multiple cores using Task manager, use the Interprocess Data Read and Interprocess Data Write blocks to send and receive data between cores.



To generate and execute C code for your SoC models, this block requires Embedded Coder Support Package for Infineon® AURIX™ TC4x Microcontrollers or Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

Ports

Input

msg — Data message from interprocess data channel
scalar

This message port receives data messages from the connected Interprocess Data Channel block. The messages process when the Task Manager block triggers the task containing the this block. For more information on messages, see “Messages”.

Dependencies

This port appears only when **Enable simulation port** parameter is enabled.

Data Types: SoCData

Output

data — Data frame read from another processor
vector

This port emits a data frame read from another processor connected via the Interprocess data Channel block.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | bus

status — Interprocess data read status
0 | 1 | 2 | 4 | 6

The status port outputs one of these values:

- 0—No errors
- 1—Data not available
- 2—Data type mismatch
- 4—Data length mismatch
- 6—Data type and Data length mismatch

Dependencies

This port appears when you disable the **Enable simulation port** parameter.

Parameters

Enable simulation port — Enable peripheral simulation port to block
on (default) | off

Select this parameter to configure the **msg** output port to enable peripheral simulation capability.

Data type — Data type of interprocess data channel
double (default) | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | bus

Enter the data type used by the interprocess data channel.

You can selectively show or hide the **Data Type Assistant** options by clicking >> button or << button. For more information, refer to “Specify Data Types Using Data Type Assistant”.

Circular FIFO depth — Number of storage buffers

1 (default) | positive integer

Buffers are circular buffers created in shared message RAM. Number of buffer is the number of elements stored at a time in the shared message RAM. These are not FIFO but will be overwritten at next instance.

Ensure that the number of buffers specified must same at both the Tx and Rx side.

Dependencies

This parameter is visible only when you disable the **Enable simulation port** parameter.

Circular FIFO element size — Size of data vector read from interprocess data channel

1 (default) | positive integer

Enter the circular FIFO element size of the data vector read from the interprocess data channel.

Channel number — Select the channel number

0 (default) | 1 | 2 | . . .

Select the channel where you want to send the data.

Dependencies

This parameter is visible only when you disable the **Enable simulation port** parameter.

Participating cores — Select the participating cores of the processor

The options vary based on the hardware board you select

Select the participating cores as per the **Hardware board** you select in the Configuration Parameters window.

Dependencies

This parameter is visible only when you disable the **Enable simulation port** parameter.

Note Ensure that **Channel number**, **Number of buffers** and **Participating cores** parameters match for the corresponding Interprocess Data Read and Interprocess Data Write blocks of the model.

Sample time — Sample time

-1 (default) | positive scalar

Enter the sample time of the block to apply to the timer-driven task subsystem.

Version History

Introduced in R2020b

See Also

Interprocess Data Channel | Interprocess Data Write

Interprocess Data Write

Send messages to another processor using interprocessor data write



Libraries:

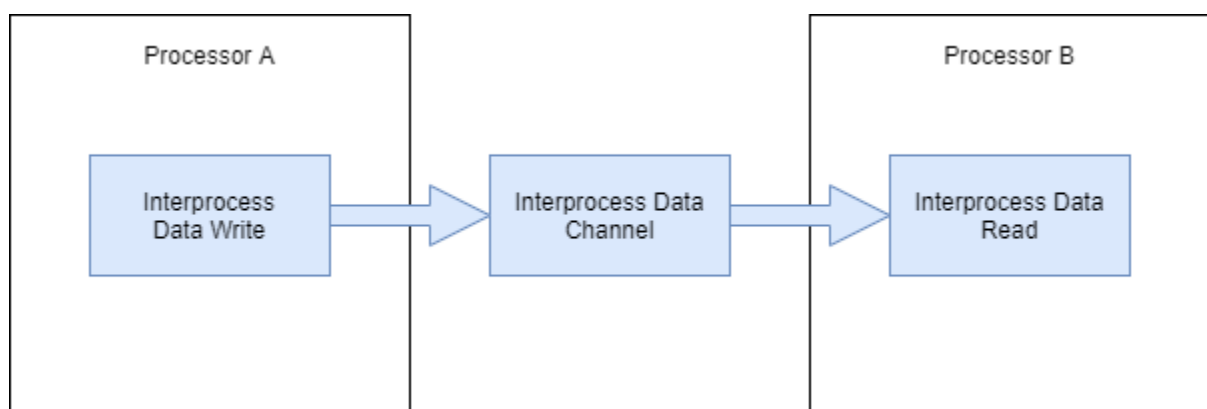
SoC Blockset / Processor Interconnect
 C2000 Microcontroller Blockset / Target Communication
 Embedded Coder Support Package for Infineon AURIX TC3x
 Microcontrollers / Target Communication
 Embedded Coder Support Package for Infineon AURIX TC4x
 Microcontrollers / Target Communication

Description

The Interprocess Data Write block asynchronously sends messages to another processor in an SoC using an interprocess data channel. The Interprocess Data Write block connects to an Interprocess Data Channel block that similarly connects to an Interprocess Data Read block contained in a separate processor reference model. In simulation, data from the current processor is asynchronously sent and processed in the processor containing the Interprocess Data Read block and associated asynchronous task. This diagram shows a generalized view of the interprocess data channel.

Note

- Ensure that the number of buffers specified must same at both the Tx and Rx side.
- If you are using multiprocessor modeling approach with single model triggering multiple cores using Task manager, use the Interprocess Data Read and Interprocess Data Write blocks to send and receive data between cores.



To generate and execute C code for your SoC models, this block requires Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers or Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

Ports

Input

data — Data input
vector

This port receives a data vector to send to another processor over the interprocess data channel.

Data Types: `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

msg — Output data message
scalar

This message port sends the output data as a message to the connected Interprocess Data Channel block. For more information on messages, see “Messages”.

Dependencies

This port appears only when **Enable simulation port** parameter is enabled.

Data Types: `SoCData`

Parameters

Enable simulation port — Enable peripheral simulation port to block
`on` (default) | `off`

Select this parameter to configure the **msg** output port to enable peripheral simulation capability.

Channels number — Select the channel number
`0` (default) | `1` | `2` | . . .

Select the channel where you want to send the data.

Dependencies

This parameter is visible only when you disable the **Enable simulation port** parameter.

Circular FIFO depth — Number of storage buffers
`1` (default) | positive integer

Buffers are circular buffers created in shared message RAM. Number of buffer is the number of elements stored at a time in the shared message RAM. These are not FIFO but will be overwritten at next instance.

Ensure that the number of buffers specified must same at both the Tx and Rx side.

Dependencies

This parameter is visible only when you disable the **Enable simulation port** parameter.

Participating cores — Select the participating cores of the processor
The options vary based on the hardware board you select

Select the participating cores as per the **Hardware board** you select in the Configuration Parameters window.

Dependencies

This parameter is visible only when you disable the **Enable simulation port** parameter.

Note Ensure that **Channel number**, **Number of buffers** and **Participating cores** parameters match for the corresponding Interprocess Data Read and Interprocess Data Write blocks of the model.

Version History

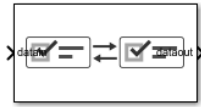
Introduced in R2020b

See Also

Interprocess Data Read | Interprocess Data Channel

Interprocess Data Channel

Model interprocessor data channel between two processors

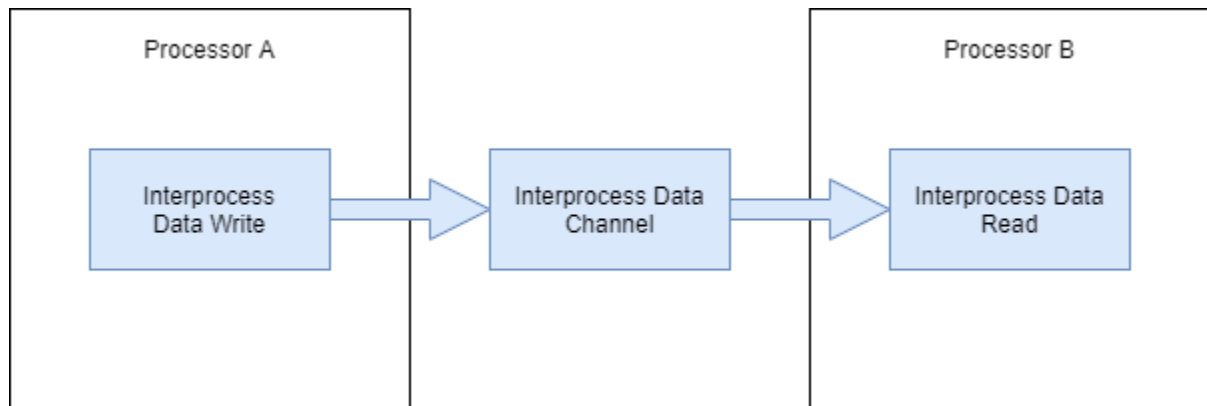


Libraries:

SoC Blockset / Processor Interconnect
C2000 Microcontroller Blockset / Test Bench Blocks
Embedded Coder Support Package for Infineon AURIX TC3x
Microcontrollers / Test Bench Blocks
Embedded Coder Support Package for Infineon AURIX TC4x
Microcontrollers / Test Bench Blocks

Description

The Interprocess Data Channel block simulates the interprocessor data channel available in multiprocessor or OS managed SoC hardware board families. The block provides a channel for asynchronous data transfer between two processors. This diagram shows a generalized view of the interprocessor data connection.



To generate and execute C code for your SoC models, this block requires Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers or Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

Note If you are using multiprocessor modeling approach with single model triggering multiple cores using Task manager, use the Interprocess Data Read and Interprocess Data Write blocks to send and receive data between cores.

Use IPC Receive and IPC Transmit blocks to receive or sent data to either core for multi-core processors. Use these blocks for flat model approach where you are using separate models for each core.

Limitations

In an SoC model, when Interprocess Data Channel blocks form a closed-loop between two or more tasks, it can create an *artificial algebraic loop* for the Simulink solver. To break the loop, the Simulink

solver implicitly adds a delay into the loop. This delay is related to an internal event and cannot be modified by the user, but the delay typically will be on the same order as the base time-step of the model. For more information on artificial algebraic loops in Simulink solvers, see “Artificial Algebraic Loops”.

Ports

Input

datain — Input data message
scalar

This message port receives input data as a message from a connected Interprocess Data Write block. For more information on messages, see “Messages”.

Data Types: SoCData

Output

dataout — Output data message
scalar

This message port sends output data as a message to a connected Interprocess Data Read block. For more information on messages, see “Messages”.

Data Types: SoCData

overwritten — Output overwrite notification signal
scalar

This port sends a true signal output whenever an overwrite of the internal buffer queue occurs. When the connected processor model executes in external mode, the connected Interprocess Data Write block generates the **overwritten** signal in the Simulation Data Inspector tool.

Dependencies

To enable this port, select the **Show when buffer is overwritten** parameter.

Data Types: Boolean

used — Output number of buffers in use
scalar

This port outputs the number of buffers currently in use in the block's internal buffer queue. When the connected processor model executes in external mode, the connected Interprocess Data Write block generates the **used** signal in the Simulation Data Inspector tool.

Dependencies

To enable this port, select the **Show number of used buffers** parameter.

Data Types: Boolean

event — Task event signal
scalar

This port sends a task event signal that triggers the Task Manager block to execute the associated event-driven task.

Dependencies

To enable this port, select the **Show event port** parameter.

Data Types: `rteEvent`

Parameters

Main

Circular FIFO depth — Number of storage buffers

1 (default) | positive integer

Buffers are circular buffers created in shared message RAM. Number of buffer is the number of elements stored at a time in the shared message RAM. These are not FIFO but will be overwritten at next instance.

Propagation delay — Propagation delay of data through the channel

1e-6 (default) | non-negative number

Specify the propagation delay of data transfers through the this block. To ignore propagation delays, set this parameter to 0.

Show event port — Option to enable task event ports

off (default) | on

Enable an event port that, when connected to the Task Manager block, can execute event-driven tasks.

Statistics

Show number of used circular FIFO — Option to enable circular FIFO count ports

off (default) | on

Enable an output port that shows the current number of circular FIFO used in the Interprocess Data Channel block internal buffer queue.

Show when circular FIFO is overwritten — Enable port that shows circular FIFO overwrites

off (default) | on

Enable an output port that signals when an overwrite of the Interprocess Data Channel block internal circular FIFO queue occurred.

Version History

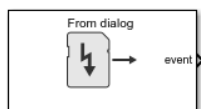
Introduced in R2020b

See Also

Interprocess Data Read | Interprocess Data Write

Event Source

Simulate and playback recorded task events



Libraries:

SoC Blockset / Processor Testbench
 C2000 Microcontroller Blockset / Scheduling
 Embedded Coder Support Package for Infineon AURIX TC3x
 Microcontrollers / Test Bench Blocks
 Embedded Coder Support Package for Infineon AURIX TC4x
 Microcontrollers / Test Bench Blocks

Description

The Event Source block, connected to the Task Manager block, enables you to simulate tasks events in your Simulink model. Task timing data can be provided from the block mask, an external file, or from an input port driven by other model signals.

Ports

Input

data — Input data
 numeric vector

Input data, specified as a numeric vector. The block converts this data into an event that corresponds to the rate of the input data. You can use this event to drive an event-driven task in the Task Manager block.

Dependencies

To enable this port, set the **Input** parameter to `From input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Output

event — Task event signal
 scalar

This port sends a task event signal that triggers the Task Manager block to execute the associated event-driven task.

Data Types: `rteEvent`

Parameters

Input — Source of input data
`From dialog (default)` | `From input port` | `From timeseries object`

Set the input data source for the block by selecting one of these values.

- **From dialog** — Input a one-dimensional array of data by using a function. Specify this function for the **Value** parameter.
- **From input port** — Input data and signals using input ports on the block.
- **From timeseries object** — Input data and time values using a timeseries object that you created in MATLAB. For more information see “Time Series Objects and Collections”.

Sample time — Time interval of sampling
-1 (default) | nonnegative scalar

Specify a discrete time interval, in seconds, at which the block outputs data.

Dependencies

To enable this parameter, set the **Input** parameter to **From dialog**.

Object name — Name of timeseries object
[] (default) | timeseries object

Specify a **timeseries** object. This **timeseries** object provides the input data for the block. For more information about time series objects, see “Time Series Objects and Collections”.

Dependencies

To enable this parameter, set the **Input** parameter to **From timeseries object**.

Version History

Introduced in R2020b

See Also

Task Manager

Task Manager

Create and manage task executions in Simulink model



Libraries:

SoC Blockset / Processor Task Execution
 C2000 Microcontroller Blockset / Scheduling
 Embedded Coder Support Package for Infineon AURIX TC3x
 Microcontrollers / Scheduling
 Embedded Coder Support Package for Infineon AURIX TC4x
 Microcontrollers / Scheduling

Description

The Task Manager block simulates the execution of software tasks as they would be expected to behave on an SoC processor. With the Task Manager, you can add and remove tasks from your model that can either be timer-driven or event-driven. Tasks can be represented in a model as rates, for timer-driven tasks, or function-call subsystems, for event-driven tasks, contained inside a single Model block. The Task Manager executes individual tasks based on their parameters, such as period, duration, trigger, priority, or processor core, and the combination of that task with the state of other tasks and their priorities in the running model.

In simulation, this block uses code generation to improve simulation speed. This entails some overhead when first simulating, but subsequent simulations are faster if the block settings remain unchanged. For more information, see “Interpreted Execution vs. Code Generation”.

Note The Task Manager block cannot be used in a referenced model. For more information on referenced models, see Model block.

The Task Manager block provides three methods to specify the duration of a task in simulation:

- A probability model of task duration defined in the block mask.
- From a data file recording of either a previous task simulation or from a task on an SoC device.
- Input ports on the block, which you can connect to more dynamic models of task duration.

Limitations

- A model containing a Task Manager blocks does not support simulation stepping. For more information on simulation stepping, see “Debug Simulations in the Simulink Editor”.

Ports

Output

Task1 — Function-call from Task1
 scalar

A function-call signal that can trigger timer-driven and event-driven tasks, represented as rate or function-call subsystems in the processor Model block, respectively.

For a rate port from a timer-driven subsystem, to show on the Model block, set the **Block Parameters > Main > Schedule rates** and select ports. For a function-call port from an event-driven subsystem contained in a Function-Call Subsystem block to show on the Model block, include an Inport in the processor Model block connected to the function-call trigger port of the subsystem. In the Inport, check **Block Parameters > Signal Attributes > Output function call**.

Note The *Task1* port must be connected to either a function-call port or scheduled rate signal port on a Model block.

Dependencies

To create or remove a control signal port for a task, add or remove the task from the Task Manager block by clicking the **Add** or **Delete** buttons in the block dialog mask.

Input

Task1Event — Message event notification
scalar

A message port that triggers the associated event-driven task. The *Task1Event* port receives the message from Event Source block. For more information on messages, see “Messages”.

Dependencies

To show a *Task1Event* port, then *Task1* must have **Type** set to Event-driven.

Data Types: `rteEvent`

Task1Dur — Task duration
positive scalar

A positive value signal that specifies the execution duration of a task at the present time. For more information on specifying task duration, see “Task Duration”.

Dependencies

To enable this port, set the **Specify task duration via** parameter to **Input** port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Parameters

Enable task simulation — Enable simulation of task duration
`on` (default) | `off`

Enable or disable the simulation of task duration. If you clear this parameter, tasks simulate using a function-call generator inheriting their period from the fundamental sample time of the model for event-driven tasks or from the dialog for timer-driven tasks.

List of tasks — List of tasks
Task1 (default)

List of the tasks generated by the Task Manager block. Each task has a set of parameters listed in the **Main** and **Simulation** tabs of the block dialog mask.

Add — Add task

button

Add a task to the Task Manager block. During deployment, each task is encapsulated as an execution thread in the generated code. The properties of the thread are taken from the **Main** parameters for that task. During simulation, the task uses a combination of the **Main** and **Simulation** parameters for that task.

Delete — Delete existing task

button

Remove a task from the Task Manager.

Dependencies

To enable this parameter, specify at least two tasks.

Use Schedule Editor ordering — Specify task priority using Schedule Editor tool

off (default) | on

Use the Schedule Editor to specify the ordering of the tasks in the SoC model. When using the Schedule Editor, task priority is automatically assigned to the tasks based on their order in the editor and the base rate priority of the processor model.

Main

Name — Name of task

Task1 (default) | character vector

Unique name of the task. The task name must only contain alphanumeric characters and underscores.

Type — Trigger type of task

Timer-driven (default) | Event-driven

Specify the task as timer-driven or event-driven. For more information on timer- and event-driven tasks, see “Timer-Driven Task” (SoC Blockset) and “Event-Driven Tasks” (SoC Blockset), respectively.

Dependencies

To enable this parameter, set Type to Timer-driven.

Period — Timer period

0.1 (default) | positive scalar

Specify the trigger time period for timer-driven tasks.

Core — Processor core to execute task

0 (default) | non-negative integer | -1

Specify the number of the processor core where a task executes. .

To have the operating system dynamically assign a core to the task, set this value to -1.

Priority — Priority of task in scheduler

10 (default) | positive integer

Specify the scheduler's priority for the event-driven task between 1 and 99. Higher priority tasks can preempt lower priority tasks, and vice versa. The task priority range is limited by the hardware attributes. For more information on task priority, see “Task Priority and Preemption” (SoC Blockset).

Dependencies

To enable this parameter, set `Type` to `Event-driven` and `Use Schedule Editor ordering` to `off`.

Drop tasks that overrun — Drop tasks that overrun

off (default) | on

Select this parameter to force tasks to drop, rather than catch up, following an overrun instance.

Note No more than 2 instances of a task can overrun execution when `Drop tasks that overrun` is set to `off`. Any additional task instances that overrun drop automatically.

Simulation**Play recorded task execution sequence** — Enable playback from file

off (default) | on

Select this parameter for the Task Manager block to play back the recorded execution data provided from the specified **File name** parameter. For more information on replaying task execution, see “Task Execution Playback Using Recorded Data”.

Specify task duration via — Source of task execution time

Dialog (default) | Input port | Record task execution statistics

Specify the source of the timing information for the task execution.

- `Dialog` - Use a normally distributed probabilistic model with **Mean**, **Deviation**, **Min**, and **Max** defined in the block dialog mask.
- `Input port` - When set from *Input port*, the block input port dynamically defines the execution duration.
- `Record task execution statistics` - Use a normally distributed probabilistic model with mean and deviation provided in file specified by **File name**.

For more information on configuring task duration, see “Task Duration”.

Task duration settings**Add** — Adds distribution

button

Adds a distribution to the set of normal distributions that generates an execution duration.

Note Only a maximum five distributions can be assigned to a single task.

Delete — Remove distribution
button

Remove a distribution from the set of normal distributions.

Percent — Likelihood of distribution
100 (default) | positive scalar

Specify the likelihood of each normal distribution. The **Percent** weighted sum of normal distributions determines the task duration likelihood.

Note The sum of **Percent** for all the distributions in a single task must equal 100.

Mean — Mean task duration in simulation
1e-06 (default) | positive scalar

Specify the mean duration of the task during simulation of the task. The simulated task duration uses a normal distribution with a specified **Mean** and **SD** parameter values as a first-order approximation of the task behavior.

SD — Standard deviation of task duration in simulation
0 (default) | positive scalar

Specify the standard deviation duration of the task during simulation of the task. The simulated task duration uses a normal distribution with a specified **Mean** and **SD** as a first-order approximation of the task behavior.

Min — Lower limit of task duration
1e-06 (default) | positive scalar

Lower limit of a task duration distribution.

Max — Upper limit of task duration
1e-06 (default) | positive scalar

Upper limit of a task duration distribution.

File name — File containing diagnostic scheduling data
filepath

The data in this file specifies the **Mean** and **SD** parameter values. When the **Play recorded task execution sequence** parameter is selected, the specified CSV file provides the explicit task execution timing. The CSV file contains the diagnostic data of the task scheduler previously recorded from the hardware board.

Dependencies

To enable this parameter, set the **Specify task duration via** parameter to Recorded task execution statistics.

Version History

Introduced in R2019a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the SoC Builder tool. To generate and execute C code for your SoC models, Embedded Coder features are required. For more information on generating code for SoC designs, see “Generate Code and Deploy Using SoC Builder” for Infineon AURIX TC3x and “Generate Code and Deploy Using SoC Builder” for Infineon AURIX TC4x microcontrollers.

The tasks in the Task Manager block execute as threads in the generated code. The task parameters in the Task Manager block specify the priority and execution core of the thread.

See Also

IO Data Source | Memory Channel

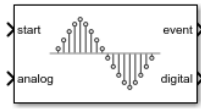
Topics

“What is Task Execution?”

“Task Duration”

ADC Interface

Convert analog signal on ADC input pin to digital signal



Libraries:

SoC Blockset / Peripherals
 C2000 Microcontroller Blockset / Test Bench Blocks
 Embedded Coder Support Package for Infineon AURIX TC3x
 Microcontrollers / Test Bench Blocks
 Embedded Coder Support Package for Infineon AURIX TC4x
 Microcontrollers / Test Bench Blocks

Description

The ADC Interface block simulates the analog-to-digital conversion (ADC) of a hardware board. The input analog signal gets sampled and converted into a representative digital value. A start event message signals the block to sample the input analog voltage signal. When the conversion completes, the block emits the digital representation of the analog signal and sends an event to a Task Manager block. At this point, a connected task can execute with the new ADC sample.

Ports

Input

start — Start analog to digital conversion
 start an analog to digital conversion event

Specify an event signal to start the sampling and measurement of the **analog** input port signal.

Data Types: `rteEvent`

analog — Analog voltage signal
 scalar

Input analog voltage signal to convert into a digital measurement.

Data Types: `double` | `single`

Output

digital — SoC message data
 scalar

This port sends the ADC Interface input signal data as a message to the **msg** input port of the ADC Read block.

Data Types: `SoCData`

wd event — Analog watchdog task event signal
 scalar

This port sends a message at whenever the analog voltage signal exceeds the specified **Lower threshold** and **Upper threshold** property values. This output connects to the input of the Task

Manager block to execute the associated event-driven task to react to the over- or under-voltage input event.

Dependencies

To enable this port, enable the **Enable analog watchdog** parameter.

Data Types: `rteEvent`

event — Task event signal
scalar

This port sends a message at each analog to digital signal conversion event. This output connects to the input of the Task Manager block to execute the associated event-driven task after executing the ADC event.

Dependencies

To enable this port, enable the **Enable interrupt** parameter.

Data Types: `rteEvent`

Parameters

Single Channel

Resolution (bits) — Resolution of digital measurement
12 (default) | 16

An input analog signal can be represented in digital values in the form of 12 or 16 bits. The minimum value of an analog signal that can be represented in 1 bit is called resolution. One bit represents the minimum voltage resolution measurable by the ADC. The minimum voltage resolution can be determined using the following equation:

where n is the **Resolution (bits)** and V_{ref} is the **Voltage reference (V)** parameter values.

Example: 16

Voltage reference (V) — Reference voltage in ADC
3 (default) | 3.3

The reference voltage determines the total voltage range that the ADC can convert into a digital value without saturating. Any voltage signal higher than this value produces the maximum possible value that can be represented by the **Resolution (bits)** parameter.

Example: 3.3

Acquisition time (s) — Time required for ADC to capture input voltage
 $320\text{e-}9$ (default) | positive scalar

Specify the time required for the ADC to capture the input voltage during sampling.

Example: $200\text{e-}9$

Conversion time (s) — Time to convert physical voltage sample to digital value
 $240\text{e-}9$ (default) | positive scalar

Specify the required time to convert the physical voltage sample to the digital representation and output the value.

Example: 20e-9

Charge/discharge time constant (s) — Charge or discharge time constant of the ADC acquisition circuit

0 (default) | nonnegative scalar

Specify the charge or discharge time constant of the ADC sample acquisition circuitry.

Multichannel

Number of channel — Number of channels used in multichannel sampling

1 (default) | integer in the range 1 to 16

Specify the number of channels used by the ADC module. Specifying 2 or more channels allows for either more efficient or precise measurements of the input signal.

Conversion type — Type multichannel conversion

Sequential (default) | Simultaneous | Oversampling

Select the type of multichannel conversion.

- **Sequential** — Take sequential measurements on each ADC channel. At a new ADC event, the next channel in the sequence of channels takes a new measurement of the input signal. All other previous channel values remain unchanged. Sequential measurement improves sampling by allowing for individual conversion times of each channel to exceed the sample rate of the ADC module.
- **Simultaneous** — Take simultaneous measurements on each ADC channel. At a new ADC event, all channels take a new measurement of the input signal, replacing the previously captured value. Simultaneous measurement allows for noise to be removed from the measurement using an average value or other filter.
- **Oversampling** — Take oversampled measurements across the channels of the ADC. Between two timer-driven ADC events, each channel takes a time offset ADC measurement, resulting in the channels sampling the input signal evenly between the two ADC events. The resulting channel output provides an oversampled measurement of the input signal at each sample. Oversampling measurement allows for the ADC module to exceed the theoretical Nyquist sample rate of the individual channel and ADC hardware.

Event

Enable interrupt — Option to enable interrupt event generation

enable (default) | disable

Select this parameter for the ADC Interface block to generate an interrupt following an ADC acquisition and to enable the **event** output port. You can connect this **event** port to a Task Manager block to simulate asynchronous ADC operation.

Condition — Condition on when to trigger interrupt

Acquisition time (default) | Acquisition + Conversion time

Select the timing condition for when to generate the ADC interrupt event. Using **Acquisition + Conversion time**, the interrupt is generated when the complete measurement is available. Using

Acquisition time, the interrupt is generated prior to the measurements availability. Allowing for the associated task to start during the conversion and reduce execution delay in the total measurement cycle.

Enable analog watchdog — Option to enable analog watchdog interrupt event generation
off (default) | on

Select this parameter for the ADC Interface block to generate an analog watchdog interrupt following an ADC acquisition where the input voltage exceeds the specified **Lower threshold** and **Upper threshold** parameter values. Selecting this parameter also enables the **wd event** output port, which you can connect to a Task Manager block to simulate task action on an over-or under-voltage event on the ADC input signal.

Lower threshold — Lower threshold watchdog trigger
0.1 (default) | real-valued scalar

Specify the lower threshold value of the analog input signal on which to trigger an analog watchdog interrupt event.

Example: 0.2

Upper threshold — Upper threshold watchdog trigger
2.9 (default) | real-valued scalar

Specify the upper threshold value of the analog input signal on which to trigger an analog watchdog interrupt event.

Example: 3.0

Interrupt latency (s) — Interrupt generation latency
0 (default) | positive scalar

Specify the time required by the ADC hardware module from the completion of the conversion to the generation of the interrupt in software.

Example: 0.00001

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the SoC Builder tool. To generate and execute C code for your SoC models, Embedded Coder features are required. For more information on generating code for SoC designs, see “Generate Code and Deploy Using SoC Builder” for Infineon AURIX TC3x and “Generate Code and Deploy Using SoC Builder” for Infineon AURIX TC4x microcontrollers.

See Also

PWM Interface | Digital IO Interface

External Websites

https://en.wikipedia.org/wiki/Analog-to-digital_converter

PWM Interface

Simulate pulse width modulation (PWM) output from hardware



Libraries:

SoC Blockset / Peripherals

C2000 Microcontroller Blockset / Test Bench Blocks

Embedded Coder Support Package for Infineon AURIX TC3x

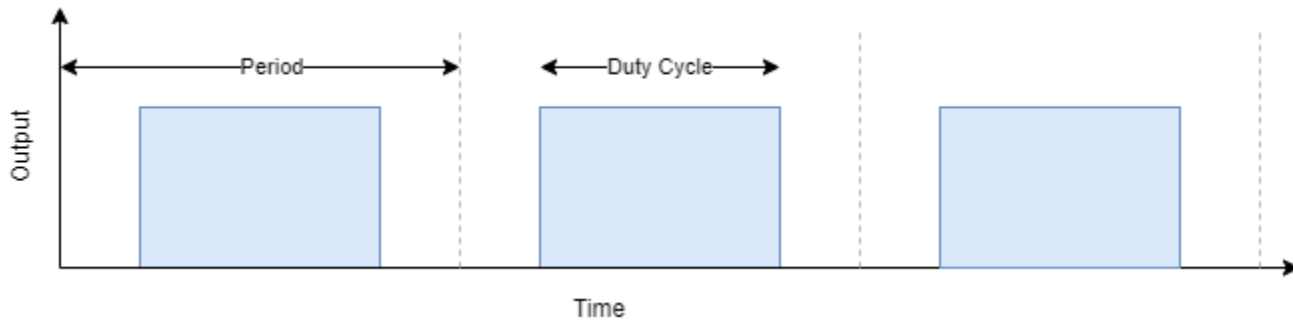
Microcontrollers / Test Bench Blocks

Embedded Coder Support Package for Infineon AURIX TC4x

Microcontrollers / Test Bench Blocks

Description

The PWM Interface block simulates the PWM output of a hardware board. This block gets duty cycle data messages from a connected PWM Write block that can either generate a switching pulse-width-modulated waveform or pass the duty cycle value to the output.



Ports

Input

msg — SoC message data
numeric vector

This port receives the duty cycle data from the **msg** port of a connected PWM Write block.

Data Types: SoCData

Output

PWM — Pulse-width-modulated signal
scalar

This port outputs the pulse-width-modulated rectangular wave defined by the **dCycle** input port.

Dependencies

To enable this port, set the **Output mode** parameter to Switching.

Data Types: double

~PWM — Complementary pulse-width-modulated signal
scalar

This port outputs the complementary **PWM** signal.

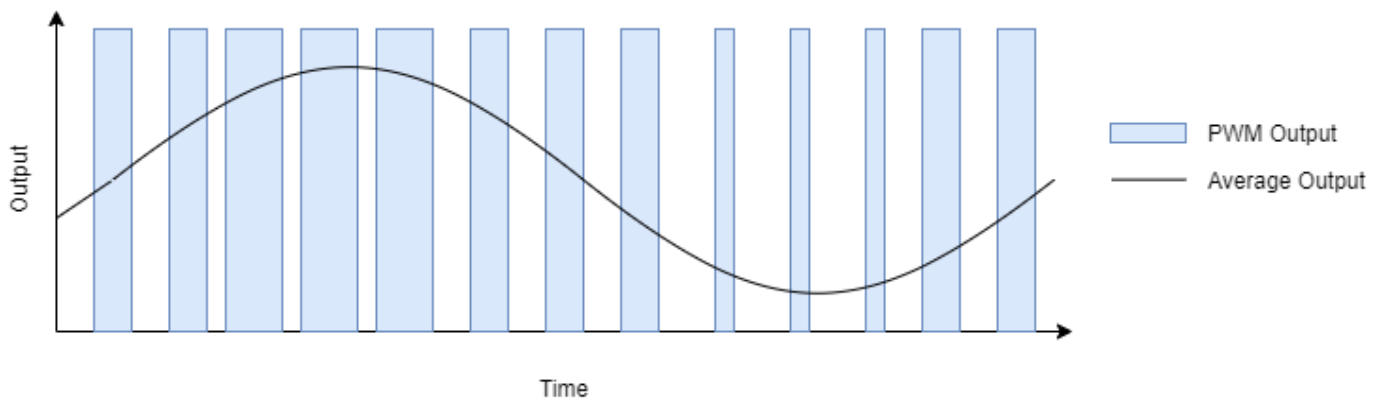
Dependencies

To enable this port, set the **Output mode** parameter to Switching.

Data Types: double

dCycle — Analog approximation of pulse-width-modulated signal
scalar

This port emits the averaged value of the PWM waveform, which is a pass-through of the duty cycle input value. This image shows the average output signal equivalent to the PWM output.



Dependencies

To enable this port, set the **Output mode** parameter to Average.

Data Types: double

event — Event emitted on each PWM cycle
scalar

This port sends a message during each PWM output event that can connect to the **start** port of the ADC Interface block to synchronize ADC and PWM events in closed-loop systems.

Dependencies

To enable this port, the **Type** parameter must be set to **ADC start** or **ADC start and PWM interrupt**.

Data Types: rteEvent

eventn — Replicate event emitted on each PWM cycle
scalar

This port creates a replica port of the **event** output port to coordinate multiple ADC modules with the PWM module.

Dependencies

To enable this port, set the **Type** parameter to `ADC_start` or `ADC_start and PWM_interrupt` and the **Number of replicas** parameter to a value greater than or equal to 2.

Data Types: `rteEvent`

interrupt — Interrupt event emitted on each PWM cycle
scalar

This port sends a message during each PWM output event that can connect to the Task Manager block to trigger other tasks in response to the PWM output update.

Dependencies

To enable this parameter, set the **Type** parameter to `PWM_interrupt` or `ADC_start and PWM_interrupt`.

Data Types: `rteEvent`

direction — Counter direction output
scalar

The block outputs the current direction as one of the following:

- 1– Down counter
- 0– Up counter

Dependencies

To enable this parameter, select the **Output counter direction** parameter.

Data Types: `Boolean`

Parameters**Main**

PWM waveform period (s) — Period of PWM waveform
`50e-6` (default) | positive scalar

Specify the period of the PWM waveform in seconds.

Note For PWM waveform period (s) of 10ns, the duty cycle must be greater than 1%.

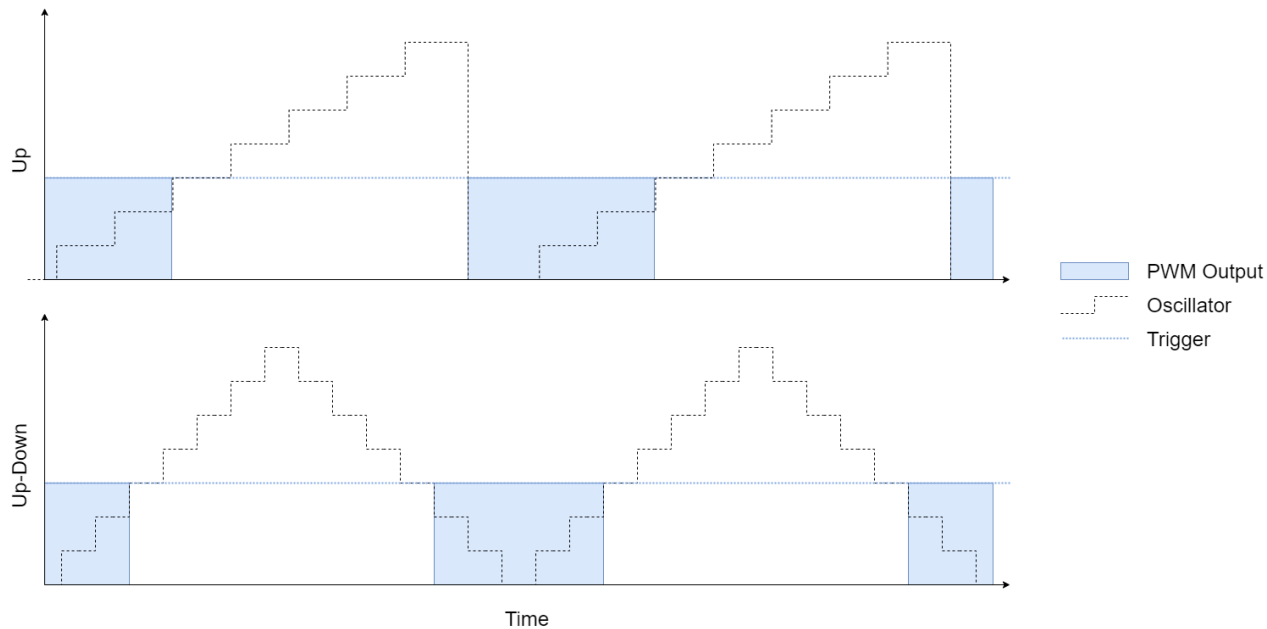
Output mode — Output mode
`Switching` (default) | `Average`

Simulate the output signal as either a true PWM waveform by specifying `Switching` or as the average of the duty cycle by specifying `Average`.

Example: `Switching`

Counter mode — Counter waveform
`Up-Down` (default) | `Up` | `Down`

The counter mode specifies the shape of the underlying sawtooth waveform that drives the PWM output signal inside the PWM module. In Up mode, the sawtooth counter increments to the maximum and then resets to zero on each period. In Down mode, the sawtooth counter decrements to zero then resets to the maximum. In Up-Down mode, the sawtooth counter oscillates from zero to the maximum value.



Example: Up

Sampling mode — Sampling mode

End of PWM period (default) | Mid or End of PWM period | Immediate (at compare matches)

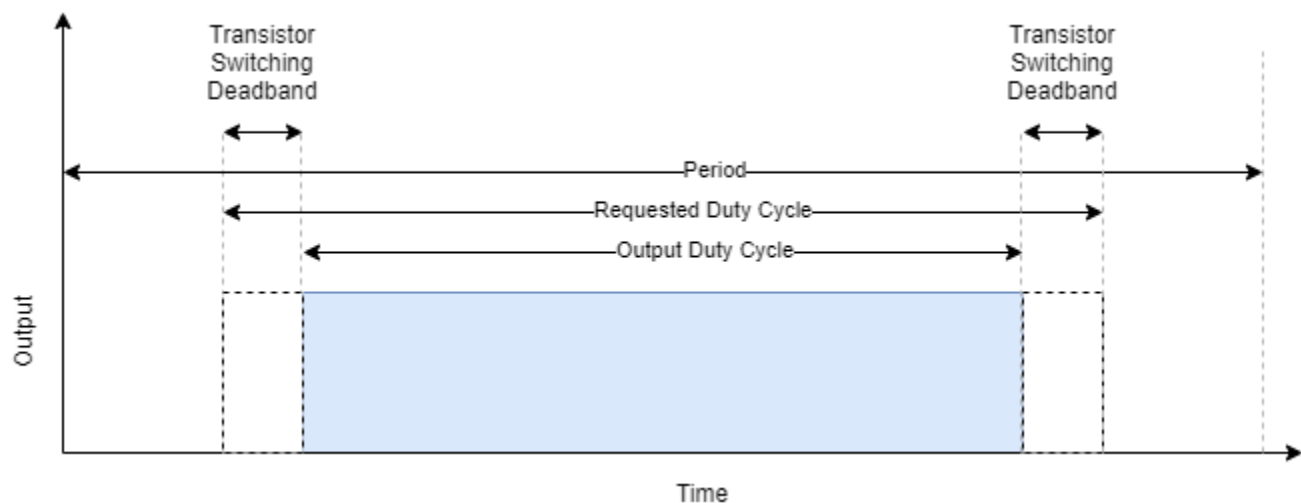
Specify the time at which the input duty cycle is sampled.

Example: Mid or End of PWM period

Dead time (s) — Dead band switching delay

1e-6 (default) | positive scalar

A time delay is introduced between turning off one of the transistors of a leg of an inverter and turning on the other transistor to ensure that a dead short circuit does not occur. This diagram shows the expected duty cycle and the delay introduced by the transistor switching the dead band.



Example: 450e-9

PWM output

At *position* of period — Signal change at position in period
High | Low | Change | NoChange

Specify the state of the PWM waveform signal at the *position* in the waveform relative to the total period. When set to High or Low, the waveform output changes to 1 or 0, respectively. When set to Change, the waveform inverts the current value. When set to NoChange, the waveform does not change. The *position* can either be the start or mid point of the PWM waveform. This table gives the default settings for these parameters.

Parameter	Default
At start of period	High
At mid of period	NoChange

Note To ensure correct PWM behaviour, set **At start of period** parameter value to Low only when the duty cycle data from the **msg** port lies in the range of (0,100).

Dependencies

At mid of period is only available when the **Counter mode** parameter is set to Up-Down.

At compare *n* — Signal change at comparator *n* trigger
High | Low | Change | NoChange

Specify the state of the PWM waveform signal when the internal PWM counter triggers comparator *n*. When set to High or Low, the waveform output changes to 1 or 0, respectively. When set to Change, the waveform inverts the current value. When set to NoChange, the waveform does not change. Two comparators, 1 and 2, are available to modify the PWM signal. This table gives the default settings for these parameters.

Parameter	Default
At compare 1	Low
At compare 2	NoChange

Dependencies

At compare 1 and **At compare 2** parameters are only available when the **Counter mode** parameter is set to Up or Down.

At compare *n* direction count — Signal change at comparator *n* trigger

High | Low | Change | NoChange

Specify the state of the PWM waveform signal when the internal PWM counter crosses the comparator *n* value in the specified *direction*. When set to **High** or **Low**, the waveform output changes to 1 or 0, respectively. When set to **Change**, the waveform inverts the current value. When set to **NoChange**, the waveform does not change. Two comparators, 1 and 2, are available to modify the PWM signal. This table gives the default settings for these parameters.

Parameter	Default
At compare 1 up count	Low
At compare 1 down count	High
At compare 2 up count	NoChange
At compare 2 down count	NoChange

Dependencies

These arguments are only available when the **Counter mode** parameter is set to Up-Down.

Output counter direction — Option to enable output port for counter direction

off | on

Select this parameter to enable the **Direction** output port in the block.

Dependencies

To enable this parameter, set the **Counter mode** parameter to Up-Down.

Phase

Phase offset in degree (0-360) — PWM waveform offset

scalar from 0 to 360

Specify the phase of the PWM waveform relative period of waveform. The phase is represented as a scalar between 0 to 360 degrees.

Event

Type — Type of events to generate

ADC start (default) | PWM interrupt | ADC start and PWM interrupt

Specify the types of events on which to generate events. When the **Type** value is set to:

- **ADC start** — Generate an event to trigger the start ADC conversion.

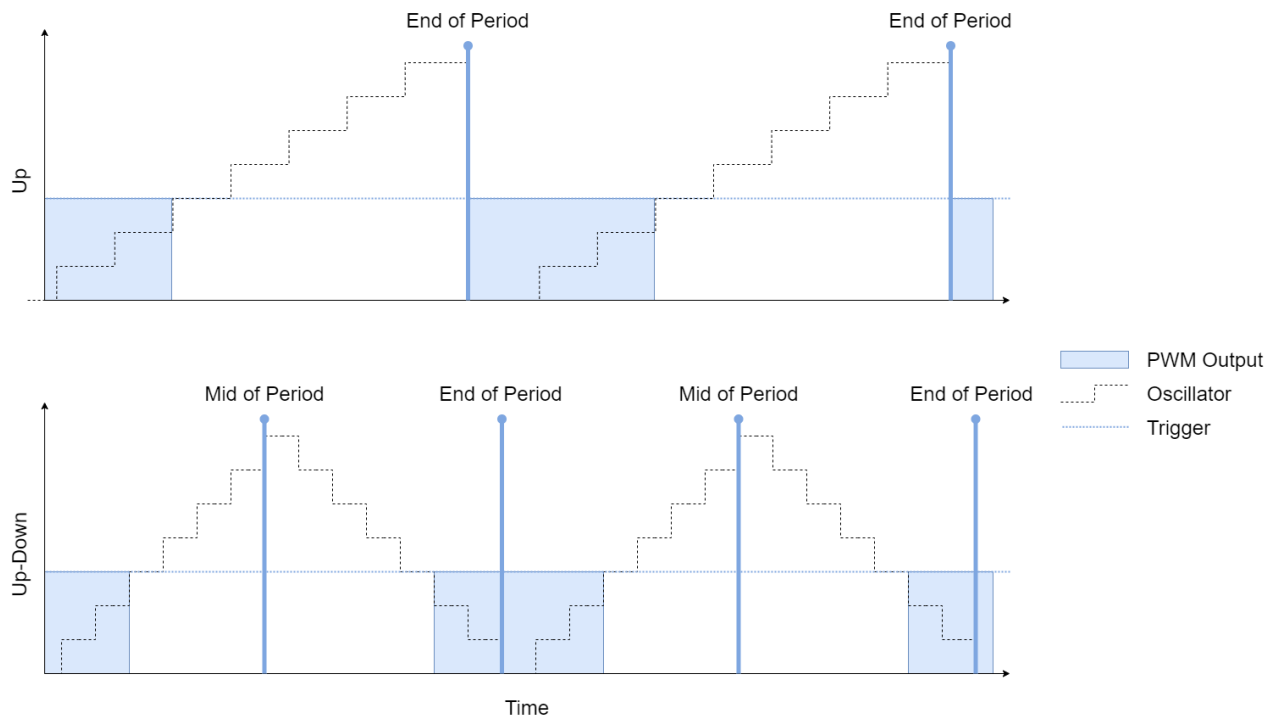
- **PWM interrupt** — Generate an interrupt event to trigger the start of a task.
- **ADC start and PWM interrupt** — Generate events for both ADCs and tasks.

Example: ADC start and PWM interrupt

ADC start condition — Trigger mode relative to PWM waveform

End of PWM period (default) | Mid of PWM period | Mid or End of PWM period | Compare 1 up count | Compare 1 down count | Compare 2 up count | Compare 2 down count

Specify when this block triggers an event relative to the PWM waveform.



Example: Mid or End of PWM period

Dependencies

To enable this parameter, the **Type** parameter must be set to **ADC start** or **ADC start and PWM interrupt**.

Generate on — Generate event on multiple of PWM update

1st event (default) | *n*th event | 16th event

Specify to generate and output an ADC trigger event on the specified multiple of the PWM vent. For example, if **Generate on** is set to the 6th event, the PWM Interface block receives 6 messages updates the output 6 times before generating an ADC event message.

Example: 4th event

Dependencies

To enable this parameter, the **Type** parameter must be set to **ADC start** or **ADC start and PWM interrupt**.

Number of replicas — Generate replica event ports

1 (default) | integer from 1 to 16

Generate replica ADC event ports and events on the block. Use this coordinate the triggering of multiple ADCs modules from a single PWM Interface block.

Example: 4

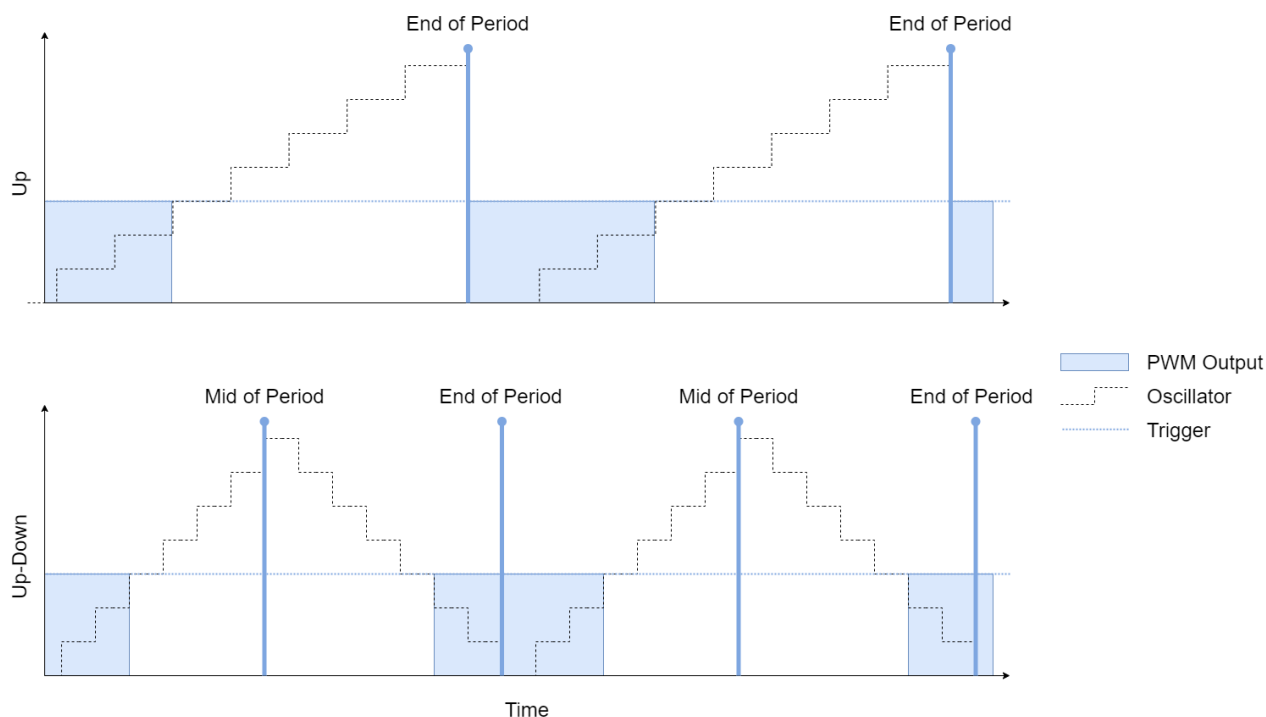
Dependencies

To enable this parameter, the **Type** parameter must be set to `ADC start` or `ADC start and PWM interrupt`.

PWM interrupt condition — Trigger mode relative to PWM waveform

End of PWM period (default) | Mid of PWM period | Mid or End of PWM period | Compare 1 up count | Compare 1 down count | Compare 2 up count | Compare 2 down count

Specify when this block triggers an interrupt event relative to the PWM waveform.



Example: Mid or End of PWM period

Dependencies

To enable this parameter, the **Type** parameter must be set to `PWM interrupt` or `ADC start and PWM interrupt`.

Interrupt latency (s) — Interrupt generation latency

0 (default) | positive number

Specify the time required by the PWM hardware module from the completion of the output update to the generation of the interrupt in software.

Example: 0.00001

Dependencies

To enable this parameter, the **Type** parameter must be set to `PWM interrupt` or `ADC start and PWM interrupt`.

Generate on — Generate event on multiple of PWM update

`1st event (default)` | `nth event` | `16th event`

Specify to generate and output a PWM interrupt trigger event on the specified multiple of the PWM vent. For example, if **Generate on** is set to the `6th event`, the PWM Interface block receives 6 messages updates the output 6 times before generating an PWM interrupt event message.

Example: `4th event`

Dependencies

To enable this parameter, the **Type** parameter must be set to `PWM interrupt` or `ADC start and PWM interrupt`.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

To automatically generate C code for your design, and execute on an SoC device, use the SoC Builder tool. To generate and execute C code for your SoC models, Embedded Coder features are required. For more information on generating code for SoC designs, see “Generate Code and Deploy Using SoC Builder” for Infineon AURIX TC3x and “Generate Code and Deploy Using SoC Builder” for Infineon AURIX TC4x microcontrollers.

See Also

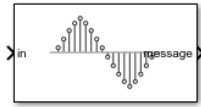
`ADC Interface` | `Digital IO Interface`

External Websites

https://en.wikipedia.org/wiki/Pulse-width_modulation

Digital IO Interface

Simulate digital input and output pins on processor



Libraries:

SoC Blockset / Peripherals
 Embedded Coder Support Package for Infineon AURIX TC3x
 Microcontrollers / Test Bench Blocks
 Embedded Coder Support Package for Infineon AURIX TC4x
 Microcontrollers / Test Bench Blocks

Description

The Digital IO Interface block simulates the behavior of a GPIO processor pin on a hardware board. You can configure the block as either an input digital block or an output digital block.

When you configure this block as an input, the block captures the crossing event of an input signal on the pin. When the crossing event occurs, the block emits the logical state of the pin as a signal and sends an event to a Task Manager block. A task can then read the logical value using an IO Data Sink block.

When you configure this block as an output, the block emits the logical signal from a connected IO Data Source block.

Ports

Input

msg — SoC message data
 scalar

This port receives the logical signal from the **msg** port of a connected IO Data Source block.

Dependencies

To enable this port, set the **IO direction** parameter to Output.

Data Types: SoCData

in — Input digital signal
 scalar

This port specifies the input digital voltage signal to convert into a digital logical measurement.

Dependencies

To enable this port, set the **IO direction** parameter to Input.

Data Types: single | double

Output

out — Output digital signal
scalar

This port outputs the logical signal value provided from the **msg** input port.

Dependencies

To enable this port, set the **IO direction** parameter to Output.

Data Types: `single` | `double`

event — Task event signal
scalar

This port sends a message at each digital IO threshold event. This output connects to the input of the Task Manager block to execute the associated event-driven task after executing the IO event.

Dependencies

To enable this port, set the **IO direction** parameter to Input and select the **Enable event** parameter.

Data Types: `rteEvent`

message — SoC message data
scalar

This port sends the Digital IO Interface input signal data as a message to the **msg** input port of the IO Data Sink block.

Dependencies

To enable this port, set the **IO direction** parameter to Input.

Data Types: `SoCData`

Parameters

IO direction — Configure block as digital input or output
Input (default) | Output

Select a value for the Digital IO Interface block to act as a digital input or a digital output.

Low input voltage — Low input voltage threshold
0.2 (default) | nonnegative scalar

Specify the lower threshold voltage of the input signal. When the signal value crosses below the threshold, the block registers this crossing as an input change, generating an event and new message with a digital value of 0.

Example: 0.3

High input voltage — High input voltage threshold
2.5 (default) | nonnegative scalar

Specify the upper threshold voltage of the input signal. When the signal value crosses above the threshold, the block registers this crossing as an input change, generating an event and new message with a digital value of 1.

Example: 0.3

Enable event — Option to enable event generation

on (default) | off

Select this parameter for the Digital IO Interface block to generate an event following a digital transition event. Selecting this parameter also enables the **event** output port, which you can connect to a Task Manager to simulate asynchronous digital read operation.

Condition — Condition on when to trigger event

Rising edge (default) | Falling edge | Both

Select the condition for when to generate the digital input event.

- **Rising edge** — Generate an event when the input signal on the **in** input port is greater than the **High input voltage** parameter value.
- **Falling edge** — Generate an event when the input signal on the **in** input port is less than the **Low input voltage** parameter value.
- **Both** — Generate an event on both conditions of the **Rising edge** and **Falling edge** options.

Sample time — Sample time

-1 (default) | positive scalar

Specify the sample time (in seconds) to define the rate at which to output the message containing the digital input data.

Version History

Introduced in R2021b

See Also

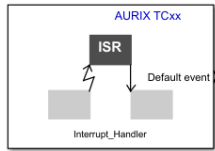
IO Data Sink | IO Data Source | Task Manager | ADC Interface | PWM Interface

External Websites

https://en.wikipedia.org/wiki/General-purpose_input/output

Hardware Interrupt

Trigger downstream function-call subsystem from an interrupt service routine



Libraries:

Embedded Coder Support Package for Infineon AURIX TC4x
Microcontrollers / AURIX TC4x

Embedded Coder Support Package for Infineon AURIX TC3x
Microcontrollers / AURIX TC3x

Description

Use the Hardware Interrupt block to create an interrupt service routine (ISR) automatically in the generated code of your model to react to the events selected. The ISR executes the downstream function-call subsystem associated with event ports of the block.

The function call subsystem associated with the event output port run at same priority as that of ISR priority.

Using this block you can:

- Create ISRs.
- Set ISR priority.
- Enable or disable interrupt preemption.

The Hardware Mapping tool allows you to configure the hardware interrupts tasks for the selected hardware board.

Ports

Input

Default event task — Function-call event input simulation

scalar

The simulation-only message input acts as pass-through with the output emitted on the block in simulation.

Dependencies

To enable this port, select the **Enable simulation port** parameter.

Output

Default event — Generate interrupt request

scalar

The output of this block is a function-call.

Parameters

Simulink task priority — Set priority of selected ISR

50 (default) | 0-255

The value you specify in this parameter sets the priority of the downstream function-call subsystem. The simulink task priority of the selected (ISR) is relative to the model base rate priority.

Run interrupt service routine as atomic unit — Select to disable interrupt preemption

off (default) | on

Select this parameter to block all other interrupts while executing the selected interrupt service routine.

Enable simulation port — Enable peripheral simulation ports in the block

off (default) | on

Select this parameter to enable peripheral simulation capability.

Port Type — Specify the type of peripheral simulation port

SoC Data (Message based) (default) | Signal

Select this parameter to configure either the SoC compatible or the Simulink signal based simulation ports to enable peripheral simulation capability.

Dependencies

To view this parameter, select the **Enable simulation port** parameter.

Version History

Introduced in R2022b

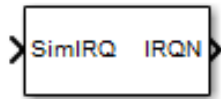
See Also

Tasks | Hardware Mapping | Tasks

Simulink Coder Blocks

Async Interrupt

Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that execute downstream subsystems or Task Sync blocks

**Libraries:**

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 Use the blocks in the Interrupt Templates block library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for a 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 | vector

A simulated interrupt source, specified as a scalar or vector.

Output

Output — Control signal
scalar | vector

Control signals for these model elements, specified as a scalar or vector:

- 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 for parameter **VME interrupt number(s)**. 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 parameter **VME interrupt number(s)**.

Parameter **Simulink task priority** values are required to generate a rate transition code (see “Asynchronous Rate Transitions”). 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 for parameter **VME interrupt number(s)**.

Note The number of elements in the arrays specifying parameters **VME interrupt vector offset(s)** and **Simulink task priority** must match the number of elements in the array specified for parameter **VME interrupt number(s)**.

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 parameter **Timer size**.

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. Parameter **Timer resolution** 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 for parameter **Timer resolution**.

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” and “Async Interrupt Block Implementation”.

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 parameter **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 parameters **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. When parameter **Timer size** is set to `auto`, you can indirectly control the word size of the counters by setting parameter **Application lifespan (days)**. 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 “Optimize Memory Usage and Prevent Overflows for Time Counters”. See also “Timers in Asynchronous Tasks”.

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 Variant Source block with the **Variant control mode** parameter set to `sim_codegen_switching`. If you use the Variant Source block, the sample times of driving blocks contribute to the sample times supported in the generated code.

Version History

Introduced in R2006a

See Also

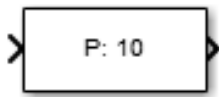
Task Sync

Topics

“Asynchronous Events”

Asynchronous Task Specification

Specify priority of asynchronous task represented by referenced model triggered by asynchronous interrupt



Libraries:
Simulink Coder / Asynchronous

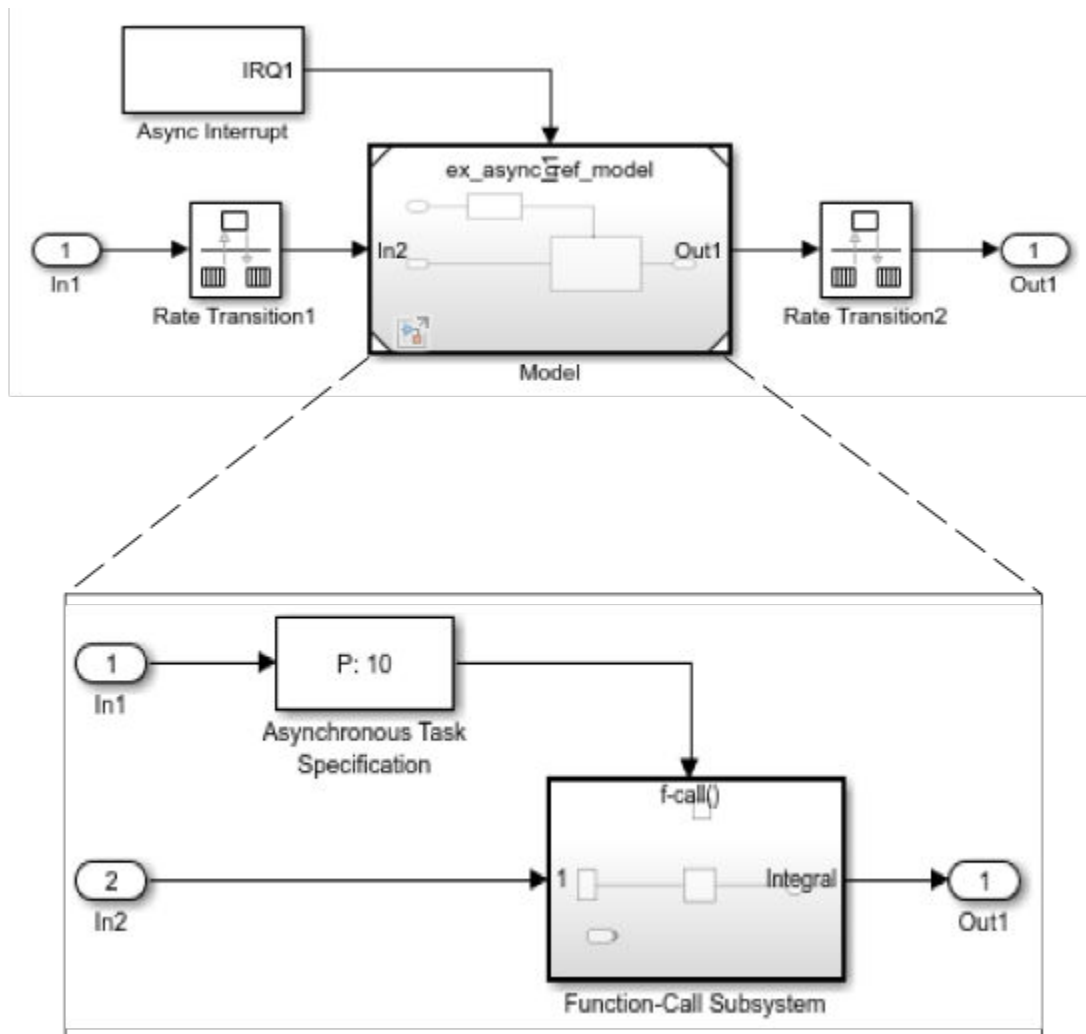
Description

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 Asynchronous Subsystem to a Referenced Model”.

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 combination of these two blocks constitutes the modeling style known as JMAAB-B.
- 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

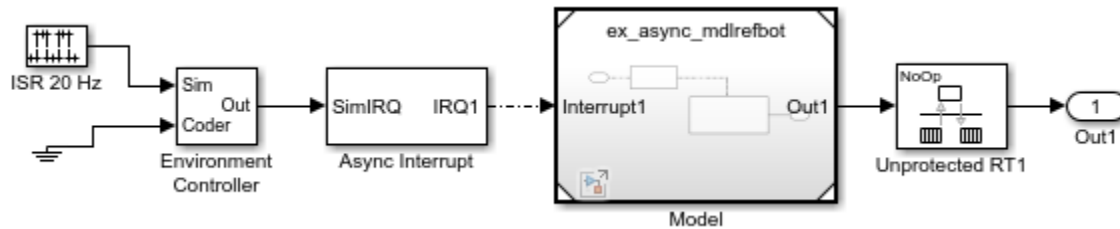
Task priority — Priority of asynchronous task that calls function-call subsystem

10 (default)

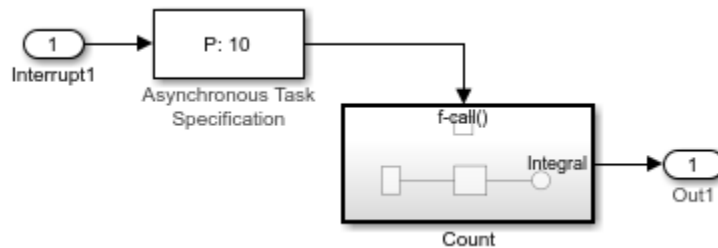
Specify an integer or [] as the priority of the asynchronous task that calls the connected function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

- 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.

Version History

Introduced in R2011a

See Also

Blocks

Inport | Function-Call Subsystem

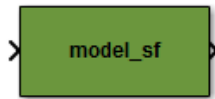
Topics

"Asynchronous Events"

"Pass Asynchronous Events in RTOS as Input to a Referenced Model"
"Convert Asynchronous Subsystem to a Referenced Model"
"Asynchronous Rate Transitions"
"Asynchronous Support"
"Asynchronous Events"
"Model References"

Generated S-Function

Represent model or subsystem as generated S-function code



Libraries:

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.

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”.
- 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 “Solver Type for Top Models with Generated S-Functions”.

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.

Version History

Introduced in R2011b

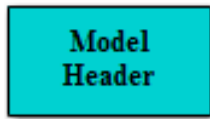
See Also

Topics

“Use S-Function Target for Model or Subsystem”

Model Header

Specify external header code

**Libraries:**

Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

Top of Model Header — Code to add near top of generated header file

no default

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

no default

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)`.

Version History

Introduced in R2006a

See Also

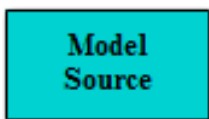
Model Source | System Derivatives | System Disable | System Enable | System Initialize | System Outputs | System Start | System Terminate | System Update

Topics

“Place External C/C++ Code in Generated Code”

Model Source

Specify external source code



Libraries:

Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

Top of Model Source — Code to add near top of generated source file

no default

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 Source — Code to add at bottom of generated source file

no default

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)`.

Version History

Introduced in R2006a

See Also

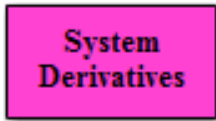
Model Header | System Derivatives | System Disable | System Enable | System Initialize | System Outputs | System Start | System Terminate | System Update

Topics

“Place External C/C++ Code in Generated Code”

System Derivatives

Specify external system derivative code



Libraries:
Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Derivatives Function Declaration Code — Code to add to the declaration section of the generated function

no default

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

no default

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

no default

Specify code that you want the code generator to add to the exit section of the `SystemDerivatives` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

[Model Header](#) | [Model Source](#) | [System Disable](#) | [System Enable](#) | [System Initialize](#) | [System Outputs](#) | [System Start](#) | [System Terminate](#) | [System Update](#)

Topics

[“Place External C/C++ Code in Generated Code”](#)

System Disable

Specify external system disable code



Libraries:
Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Disable Function Declaration Code — Code to add to the declaration section of the generated function
no default

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
no default

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
no default

Specify code that you want the code generator to add to the exit section of the `SystemDisable` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

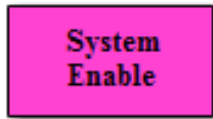
[Model Header](#) | [Model Source](#) | [System Derivatives](#) | [System Enable](#) | [System Initialize](#) | [System Outputs](#) | [System Start](#) | [System Terminate](#) | [System Update](#)

Topics

[“Place External C/C++ Code in Generated Code”](#)

System Enable

Specify external system enable code



Libraries:
Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Enable Function Declaration Code — Code to add to the declaration section of the generated function

no default

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

no default

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

no default

Specify code that you want the code generator to add to the exit section of the `SystemEnable` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

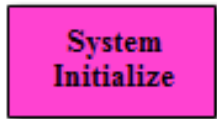
[Model Header](#) | [Model Source](#) | [System Derivatives](#) | [System Disable](#) | [System Initialize](#) | [System Outputs](#) | [System Start](#) | [System Terminate](#) | [System Update](#)

Topics

[“Place External C/C++ Code in Generated Code”](#)

System Initialize

Specify external system initialization code

**Libraries:**

Simulink Coder / Custom 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. If the model also contains an Initialize Function, the code generator places generated code for that block after the code that you specify in the System Initialize block.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Initialize Function Declaration Code — Code to add to the declaration section of the generated function
no default

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
no default

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
no default

Specify code that you want the code generator to add to the exit section of the `SystemInitialize` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

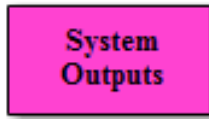
[Model Header](#) | [Model Source](#) | [System Derivatives](#) | [System Disable](#) | [System Enable](#) | [System Outputs](#) | [System Start](#) | [System Terminate](#) | [System Update](#)

Topics

["Place External C/C++ Code in Generated Code"](#)

System Outputs

Specify external system outputs code



Libraries:
Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Outputs Function Declaration Code — Code to add to the declaration section of the generated function

no default

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

no default

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

no default

Specify code that you want the code generator to add to the exit section of the `SystemOutputs` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

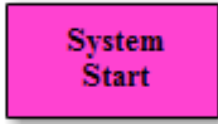
[Model Header](#) | [Model Source](#) | [System Derivatives](#) | [System Disable](#) | [System Enable](#) | [System Initialize](#) | [System Start](#) | [System Terminate](#) | [System Update](#)

Topics

[“Place External C/C++ Code in Generated Code”](#)

System Start

Specify external system startup code



Libraries:
Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Start Function Declaration Code — Code to add to the declaration section of the generated function
no default

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
no default

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
no default

Specify code that you want the code generator to add to the exit section of the `SystemStart` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

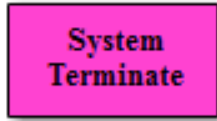
Model Header | Model Source | System Derivatives | System Disable | System Enable | System Initialize | System Outputs | System Terminate | System Update

Topics

“Place External C/C++ Code in Generated Code”

System Terminate

Specify external system termination code

**Libraries:**

Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Terminate Function Declaration Code — Code to add to the declaration section of the generated function

no default

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

no default

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

no default

Specify code that you want the code generator to add to the exit section of the `SystemTerminate` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

Model Header | Model Source | System Derivatives | System Disable | System Enable | System Initialize | System Outputs | System Start | System Update

Topics

“Place External C/C++ Code in Generated Code”

System Update

Specify external system update code



Libraries:
Simulink Coder / Custom 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.

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.

Note This block is intended for inserting arbitrary custom code into the generated code in specific locations. See the C Caller and C Function blocks, which are more typically used for integrating custom algorithmic code into a model.

Parameters

System Update Function Declaration Code — Code to add to the declaration section of the generated function

no default

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

no default

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

no default

Specify code that you want the code generator to add to the exit section of the `SystemUpdate` function for the model or subsystem.

Version History

Introduced in R2006a

See Also

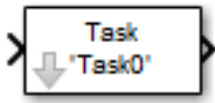
Model Header | Model Source | System Derivatives | System Disable | System Enable | System Initialize | System Outputs | System Start | System Terminate

Topics

“Place External C/C++ Code in Generated Code”

Task Sync

Run code of downstream function-call subsystem or Stateflow chart by spawning an example RTOS (VxWorks) task

**Libraries:**

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 Use the blocks in the Interrupt Templates block library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for a 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”). 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 parameter appears only if **Synchronize the data transfer of this task with the caller task** is cleared. 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 model configuration parameter **Application lifespan (days)** and block parameter **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)** parameter. 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 “Optimize Memory Usage and Prevent Overflows for Time Counters”. See also “Timers in Asynchronous Tasks”.

Version History

Introduced in R2006a

See Also

Async Interrupt

Topics

“Asynchronous Events”

Embedded Coder Parameters: Advanced Parameters

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
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”.

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” 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”

Feature set for selected hardware board

Select to use features from either the SoC Blockset™ support package or a Simulink Coder or Embedded Coder support package. Features, including properties and blocks, for the selected hardware board in SoC Blockset cannot be used in the Simulink Coder or Embedded Coder hardware support package.

Note This selection only appears when the SoC Blockset and a Simulink Coder or Embedded Coder support package for the same hardware board are both installed.

Settings

Default: SoC Blockset, Simulink or Embedded Coder Hardware Support Package

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. Based on your system target file selection, `autosar.tlc` or `autosar_adaptive.tlc`, you see code generation parameters for the AUTOSAR Classic Platform or the AUTOSAR Adaptive Platform.

Classic Platform Parameter	Description
Generate XML file for schema version	Select the AUTOSAR Classic Platform schema version to use when generating XML files.
Maximum SHORT-NAME length	Specify maximum length for SHORT-NAME XML elements for the AUTOSAR Classic Platform.
Use AUTOSAR compiler abstraction macros	Specify use of AUTOSAR macros to abstract compiler directives.
Support root-level matrix I/O using one-dimensional arrays	Allow root-level matrix I/O for column-major array layout.

Adaptive Platform Parameter	Description
Generate XML file for schema version	Select the AUTOSAR Adaptive Platform schema version to use when generating XML files.
Maximum SHORT-NAME length	Specify maximum length for SHORT-NAME XML elements for the AUTOSAR Adaptive Platform.
Transport layer	Select transport layer for XCP Server communication.
IP address	Specify IP address for XCP communication.
Port	Specify network port for XCP communication.
Verbose	Enable verbose XCP server messages.
Use custom XCP Server	Specify whether to use custom or default XCP server.

See Also

More About

- “Code Generation” (AUTOSAR Blockset)
- “Code Generation” (AUTOSAR Blockset)
- “Model Configuration Set Customization”

Code Generation: AUTOSAR Code Generation Options Tab Overview

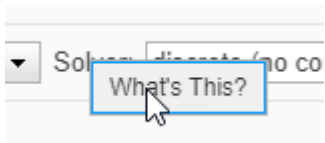
Parameters for controlling AUTOSAR code generation.

Configuration

This pane appears only if you specify the `autosar.tlc` or `autosar_adaptive.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

To configure other AUTOSAR options, use the AUTOSAR code perspective. From the **Apps** tab, open the AUTOSAR Component Designer app.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

Generate XML file for schema version

Generate AUTOSAR Classic XML files for specified schema version

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Select the AUTOSAR Classic Platform schema version to use when generating XML files. All models in a model hierarchy must specify the same value for this parameter.

Dependencies

To enable this parameter, set the **SystemTargetFile** parameter to `autosar.tlc`.

Settings

R23-11 (default) | R22-11 | R21-11 | R20-11 | R19-11 | 4.4 | 4.3 | 4.2 | 4.1 | 4.0

- R23-11 — Use schema version R23-11
- R22-11 — Use schema version R22-11
- R21-11 — Use schema version R21-11
- R20-11 — Use schema version R20-11
- R19-11 — Use schema version R19-11
- 4.4 — Use schema version 4.4 (revision 4.4.0)
- 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)

Tips

- Selecting the AUTOSAR target for your model for the first time sets the schema version parameter to the default value, R23-11.
- 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 AUTOSAR Classic Schema” (AUTOSAR Blockset).
- To configure other AUTOSAR options, use the AUTOSAR code perspective. From the **Apps** tab, open the AUTOSAR Component Designer app.

Recommended Settings

Application	Setting
Debugging	No Impact

Application	Setting
Traceability	No Impact
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AutosarSchemaVersion

Type: character vector

Value: 'R23-11' | 'R22-11' | 'R21-11' | 'R20-11' | 'R19-11' | '4.4' | '4.3' | '4.2' | '4.1' | '4.0'

Default: 'R23-11'

Version History

Introduced in R2011a

R2025a: Use schema version R23-11 with AUTOSAR Classic Platform

R2025a extends support of the AUTOSAR Classic Platform to include release R23-11. AUTOSAR Blockset supports schema version R23-11 for import and export of ARXML files and generation of AUTOSAR-compliant C code.

If you import schema R23-11 files into Simulink, the ARXML importer detects, uses, and sets the schema version of the parameter in the model. If you create an AUTOSAR model in Simulink, the default schema is R23-11.

R2024a: Use schema version R22-11 with AUTOSAR Classic Platform

R2024a extends support of the AUTOSAR Classic Platform to include release R22-11. AUTOSAR Blockset supports schema version R22-11 for import and export of ARXML files and generation of AUTOSAR-compliant C code.

If you import schema R22-11 files into Simulink, the ARXML importer detects, uses, and sets the schema version of the parameter in the model. If you create an AUTOSAR model in Simulink, the default schema is R22-11.

R2023a: Use schema version R21-11 with AUTOSAR Classic Platform

R2023a extends support of the AUTOSAR Classic Platform to include release R21-11. AUTOSAR Blockset supports schema version R21-11 for import and export of ARXML files and generation of AUTOSAR-compliant C code.

If you import schema R21-11 ARXML files into Simulink, the ARXML importer detects and uses the schema version and sets the schema version parameter in the model. If you create an AUTOSAR model in Simulink the default schema is R21-11.

R2022b: Use schema version R20-11 with AUTOSAR Classic Platform

R2022b extends support of the AUTOSAR Classic Platform to include release R20-11. AUTOSAR Blockset supports schema version R20-11 for import and export of ARXML files and generation of AUTOSAR-compliant C code.

If you import schema R20-11 ARXML files into Simulink the ARXML importer detects and uses the schema version and sets the schema version parameter in the model. If you create an AUTOSAR model in Simulink, the default schema version is R20-11.

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Select AUTOSAR Classic Schema” (AUTOSAR Blockset)

“Code Generation” (AUTOSAR Blockset)

Maximum SHORT-NAME length

Maximum length for SHORT -NAME elements generated in XML files

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Set maximum length for SHORT -NAME XML elements for the AUTOSAR Classic Platform

Settings

128 (default) | integer

The AUTOSAR standard specifies that the maximum length of SHORT -NAME XML elements is 128 characters. To configure a maximum length for SHORT -NAME elements exported by the code generator, specify an integer value between 32 and 128, inclusive.

Tip

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Impact
Traceability	No Impact
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AutosarMaxShortNameLength

Type: integer

Value: integer value between 32 and 128, inclusive

Default: 128

Version History

Introduced in R2011a

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Specify Maximum SHORT-NAME Length” (AUTOSAR Blockset)

Use AUTOSAR compiler abstraction macros

Use compiler abstraction macros

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Specify use of AUTOSAR macros to abstract compiler directives

Settings

'off' (default) | 'on'

- '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

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Impact
Traceability	No Impact
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AutosarCompilerAbstraction

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2011a

See Also

Topics

“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

Use one-dimensional C arrays to support root-level matrix I/O

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Allow root-level matrix I/O for column-major array layout

Settings

'off' (default) | 'on'

- 'off' — For column-major array layout, 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.
- 'on' — For AUTOSAR component models that use column-major array layout, software supports matrix I/O at the root level by generating code that implements matrices as one-dimensional arrays.

Tips

- All models in a model hierarchy must specify the same value for this parameter.
- For an AUTOSAR component model with multidimensional arrays, if you set the model configuration parameter **Array layout** to Row-major, you can preserve dimensions of multidimensional arrays in the generated C code. Preserving array dimensions in the generated code can enhance code integration.

When **Array layout** is set to Row-major, **Support root-level matrix I/O using one-dimensional arrays** does not apply.

- Models configured for the AUTOSAR Classic Platform do not support multi-dimensional arrays for outputs mapped with data access mode **ImplicitSendByRef**. When you validate the code mappings or try to generate code, the software produces an error if outputs mapped with data access mode **ImplicitSendByRef** are present in the model. Use one-dimensional arrays or select a different data access mode.

Recommended Settings

Application	Setting
Debugging	No Impact
Traceability	No Impact
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AutosarMatrixIOAsArray

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2011a

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Root-Level Matrix I/O” (AUTOSAR Blockset)

Generate XML file for schema version

Generate AUTOSAR Adaptive XML files for specified schema version

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Note Starting in R2024b, schema versions R19-03 (00047) and R18-10 (00046) are no longer supported.

Description

Select the AUTOSAR Adaptive Platform schema version to use when generating XML files.

Dependencies

To enable this parameter, set the **SystemTargetFile** parameter to `autosar_adaptive.tlc`.

Settings

R22-11 (00051) | R21-11 (00050) (default) | R20-11 (00049) | R19-11 (00048)

- R22-11 (00051) — Use schema release R22-11 (Version 00051)
- R21-11 (00050) — Use schema release R21-11 (Version 00050)
- R20-11 (00049) — Use schema release R20-11 (Version 00049)
- R19-11 (00048) — Use schema release R19-11 (Version 00048)

Tips

- Selecting the AUTOSAR adaptive target for your model for the first time sets the schema version parameter to the default value R22-11 (00051).
- When you import ARXML code into Simulink, the ARXML importer detects the schema version and sets the schema version parameter in the model.
- All models in a model hierarchy must specify the same value for this parameter.
- To configure other AUTOSAR options, use the AUTOSAR code perspective. From the **Apps** tab, open the AUTOSAR Component Designer app.

Recommended Settings

Application	Setting
Debugging	No Impact
Traceability	No Impact
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AutosarSchemaVersion

Type: character vector

Value: 'R22-11' | 'R21-11' | 'R20-11' | 'R19-11'

Default: 'R22-11'

Version History

Introduced in R2011a

R2024a: Use schema version R22-11 with AUTOSAR Adaptive Platform

R2024a extends support of the AUTOSAR Adaptive Platform to include release R22-11. AUTOSAR Blockset supports schema version R22-11 for import and export of ARXML files and generation of AUTOSAR-compliant C code.

If you import schema R22-11 files into Simulink, the ARXML importer detects and uses the schema version and set the schema version of the parameter in the model. If you create an AUTOSAR model in Simulink, the default schema is R22-11.

R2023a: Use schema version R21-11 with AUTOSAR Adaptive Platform

R2023a extends support of the AUTOSAR Adaptive Platform to include release R21-11. AUTOSAR Blockset supports schema version R21-11 for import and export of ARXML files and generation of AUTOSAR-compliant C code.

If you import schema R21-11 ARXML files into Simulink, the ARXML importer detects and uses the schema version and sets the schema version parameter in the model. If you create an AUTOSAR model in Simulink the default schema is R21-11.

R2022b: Use schema version R20-11 with AUTOSAR Adaptive Platform

R2022b extends support of the AUTOSAR Adaptive Platform to include release R20-11. AUTOSAR Blockset supports schema version R20-11 for import and export of ARXML files and generation of AUTOSAR-compliant C code.

If you import schema R20-11 ARXML files into Simulink the ARXML importer detects and uses the schema version and sets the schema version parameter in the model. If you create and AUTOSAR model in Simulink, the default schema version is R20-11.

R2022b: Command-line argument value is in AUTOSAR platform release format

Behavior changed in R2022b

Starting in R2022b, when you use a function call to access the AutosarSchemaVersion parameter, its **Value** is now in the format of the AUTOSAR platform release 'Ryy-mm'. For example, 'R20-11', 'R19-11', 'R19-03', and 'R18-10'.

Previously the **Value** was in the form of the AUTOSAR schema version '000xx'.

R2024b: Removed

Errors starting in R2024b

Starting in R2024b schema versions R19-03 (version 00047) and R18-10 (00046) are no longer supported for the AUTOSAR Adaptive Platform.

R2024a: Warns

Warns starting in R2024a

Schema versions R19-03 (version 00047) and R18-10 (00046) will no longer be supported for the AUTOSAR Adaptive Platform in a future release. Starting in R2024a the software will warn if schema versions R19-03 (version 00047) or R18-10 (00046) are selected.

R2023b: To be removed

Not recommended starting in R2023b

Schema versions R19-03 (version 00047) and R18-10 (00046) will no longer be supported for the AUTOSAR Adaptive Platform in a future release.

See Also**Topics**

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Select AUTOSAR Adaptive Schema” (AUTOSAR Blockset)

“Code Generation” (AUTOSAR Blockset)

Maximum SHORT-NAME length

Maximum length for SHORT -NAME elements generated in XML files

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Set maximum length for SHORT -NAME XML elements generated in XML files for the AUTOSAR Adaptive Platform

Settings

128 (default) | integer

The AUTOSAR standard specifies that the maximum length of SHORT -NAME XML elements is 128 characters. To configure a maximum length for SHORT -NAME elements exported by the code generator, specify an integer value between 32 and 128, inclusive.

Tip

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Impact
Traceability	No Impact
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AutosarMaxShortNameLength

Type: integer

Value: integer value between 32 and 128, inclusive

Default: 128

Version History

Introduced in R2011a

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Specify Maximum SHORT-NAME Length” (AUTOSAR Blockset)

Transport layer

Transport layer used by XCP

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Select transport layer for XCP Server communication

Settings

'None' (default) | 'XCP On TCP/IP'

- 'None' — XCP transport layer information is not provided.
- 'XCP On TCP/IP' — Use the TCP/IP transport layer for XCP communication and enable additional parameters for specifying interface details.

Tip

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Recommendation
Traceability	No Recommendation
Efficiency	No Recommendation
Safety precaution	No Recommendation

Programmatic Use

Parameter: AdaptiveAutosarXCPServerTransportLayer

Type: character vector

Value: 'None' | 'XCP On TCP/IP'

Default: 'None'

Version History

Introduced in R2020a

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Specify XCP Server Transport Layer” (AUTOSAR Blockset)

“Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” (AUTOSAR Blockset)

IP address

IP address of the machine on which the AUTOSAR adaptive application (XCP Server) executes

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Specify IP address for XCP communication

Settings

'127.0.01' (default) | character vector

Set the IP address of the machine on which the AUTOSAR adaptive application (XCP Server) executes.

Tip

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Impact
Traceability	No Impact
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AdaptiveAutosarXCPServerTCPIPAddress

Type: character vector

Value: valid IP address

Default: 127.0.0.1

Version History

Introduced in R2020a

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Specify XCP Server IP Address” (AUTOSAR Blockset)

“Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” (AUTOSAR Blockset)

Port

Network port on which the AUTOSAR adaptive application (XCP Server) serves XCP Client commands

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Specify network port for XCP communication

Settings

'17725' (default) | character vector

Network port on which the AUTOSAR adaptive application (XCP Server) serves XCP Client commands.

Tip

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Recommendation
Traceability	No Recommendation
Efficiency	No Recommendation
Safety precaution	No Recommendation

Programmatic Use

Parameter: AdaptiveAutosarXCPServerPort

Type: character vector

Value: valid port number

Default: 17725

Version History

Introduced in R2020a

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Specify XCP Server Port” (AUTOSAR Blockset)

“Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” (AUTOSAR Blockset)

Verbose

Enable verbose messages for XCP Server

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Enable verbose XCP Server messages

Settings

'on' (default) | 'off'

- 'off' — Software generates code with verbose messages.
- 'on' — Software generates code that does *not* contain verbose messages.

Tip

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Recommendation
Traceability	No Recommendation
Efficiency	No Recommendation
Safety precaution	No Recommendation

Programmatic Use

Parameter: AdaptiveAutosarXCPServerVerbosity

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2020a

See Also

Topics

“Model Configuration Parameters: Code Generation AUTOSAR” on page 6-2

“Enable XCP Server Message Verbosity” (AUTOSAR Blockset)

“Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” (AUTOSAR Blockset)

Use custom XCP Server

Use custom or default XCP Server

Model Configuration Pane: Code Generation > AUTOSAR Code Generation Options

Description

Specify whether to use custom or default XCP Server

Enable to use custom XCP Server. Disable to use MathWorks XCP Server

Settings

'off' (default) | 'on'

- 'off' — Use a custom XCP Server, with an implementation for XCP functions provided in the XCP header file `xcp_server.h`.
- 'on' — Use the default MathWorks XCP Server.

Tip

All models in a model hierarchy must specify the same value for this parameter.

Recommended Settings

Application	Setting
Debugging	No Recommendation
Traceability	No Recommendation
Efficiency	No Recommendation
Safety precaution	No Recommendation

Programmatic Use

Parameter: AdaptiveAutosarUseCustomXCPServer

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2020a

See Also

Topics

"Model Configuration Parameters: Code Generation AUTOSAR" on page 6-2

"Use Custom XCP Server" (AUTOSAR Blockset)

“Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” (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 **Code Generation > Code Placement** pane.

Parameter	Description
Data definition	Specify where to place definitions of global variables.
Data definition filename	Specify the name of the file that is to contain data definitions.
Data declaration	Specify where <code>extern</code> , <code>typedef</code> , and <code>#define</code> statements are to be declared.
Data declaration filename	Specify the name of the file that is to contain data declarations.
#include file delimiter	Specify the type of <code>#include</code> file delimiter to use in generated code.
Use owner from data object for data definition placement	Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.
Signal display level	Specify the persistence level for MPT signal data objects.
Parameter tune level	Specify the persistence level for MPT parameter data objects.
File packaging format	Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files.
Header files	Specify customized name for generated header files.
Source files	Specify customized name for generated source files.
Data files	Specify customized name for generated data files.
Rate Transition block code	Specify the format for Rate Transition block code and data.

See Also

More About

- “Model Configuration Set Customization”

Code Generation: Code Placement Tab Overview

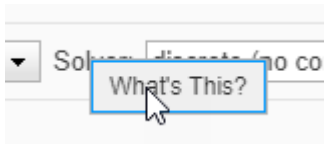
Specify the data placement in the generated code.

Configuration

This tab appears only if you specify an ERT-based System target file.

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

Placement of global variable definitions

Model Configuration Pane: Code Generation / Code Placement

Description

Specify where to place definitions of global variables.

Dependencies

This parameter enables **Data definition filename**.

Settings

Auto (default) | Data defined in source file | Data defined in a single separate source file

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.

Limitations

This parameter applies to data with storage classes except these:

- ExportedGlobal
- ImportedExtern
- ImportedExternPointer
- BitField
- FileScope
- Struct
- CompilerFlag
- Localizable (non-global variable)

In the Embedded Coder Dictionary, the **Header File** value for unsupported storage classes is empty. You cannot specify default file placement for unsupported storage classes.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: GlobalDataDefinition

Type: character vector

Value: 'Auto' | 'InSourceFile' | 'InSeparateSourceFile'

Default: 'Auto'

Version History

Introduced before R2006a

See Also

Data definition filename

Topics

“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

File to contain data definitions

Model Configuration Pane: Code Generation / Code Placement

Description

Specify the name of the file that is to contain data definitions.

Dependencies

This parameter is enabled by **Data definition**.

Settings

`global.c` (default) | C file name

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid file
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: `DataDefinitionFile`

Type: character vector

Value: a valid file

Default: `'global.c'`

Version History

Introduced before R2006a

See Also

Data definition

Topics

"Model Configuration Parameters: Code Generation Code Placement" on page 7-2

Selecting and Defining Templates

Custom File Processing

Data declaration

Location for data declarations

Model Configuration Pane: Code Generation / Code Placement

Description

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

Dependencies

This parameter enables **Data declaration filename**.

Settings

Auto (default) | Data declared in source file | Data defined in a single separate source file

Default: Auto

Auto

Lets the code generator determine where the declarations should be located.

Data declared in source file

Places declarations in `.h` header files where functions are located. The data header template file is not used. The code generator places the declarations in one or more function `.h` 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.

Limitations

This parameter applies to data with storage classes except these:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`
- `BitField`
- `FileScope`
- `Struct`
- `CompilerFlag`
- `Localizable` (non-global variable)

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: GlobalDataReference

Type: character vector

Value: 'Auto' | 'InSourceFile' | 'InSeparateHeaderFile'

Default: 'Auto'

Version History

Introduced in R2006a

See Also

Data declaration filename

Topics

“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

File to contain data declarations

Model Configuration Pane: Code Generation / Code Placement

Description

Specify the name of the file that is to contain data declarations.

Dependencies

This parameter is enabled by **Data declaration**.

Settings

`global.h` (default) | header file name

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid file
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: `DataReferenceFile`

Type: character vector

Value: a valid file

Default: `'global.h'`

Version History

Introduced before R2006a

See Also

Data declaration

Topics

“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

Whether to use data ownership setting for definition in code

Model Configuration Pane: Code Generation / Code Placement

Description

Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.

Settings

off (default) | on

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: EnableDataOwnership

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2011b

See Also

Topics

“Model Configuration Parameters: Code Generation Code Placement” on page 7-2

#include file delimiter

Type of delimiter for include file

Model Configuration Pane: Code Generation / Code Placement

Description

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

Dependencies

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

This parameter does not control the delimiter format generated from storage classes in an Embedded Coder Dictionary. This parameter does control the delimiter format for code from storage classes that you created using the custom storage class designer.

Settings

Auto (default) | #include "header.h" | #include <header.h>

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: IncludeFileDelimiter

Type: character vector

Value: 'Auto' | 'UseQuote' | 'UseBracket'

Default: 'Auto'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Code Placement” on page 7-2

Signal display level

Persistence level for MPT signal data objects

Model Configuration Pane: Code Generation / Code Placement

Description

Specify the persistence level for MPT signal data objects.

Dependencies

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

Settings

10 (default) | integer value

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 mpt.Signal properties dialog.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid integer
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: SignalDisplayLevel

Type: integer

Value: a valid integer

Default: 10

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Code Placement” on page 7-2
Selecting Persistence Level for Signals and Parameters

Parameter tune level

Persistence level for MPT parameter data objects

Model Configuration Pane: Code Generation / Code Placement

Description

Specify the persistence level for MPT parameter data objects.

Dependencies

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

Settings

10 (default) | integer value

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 mpt.Parameter properties dialog.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid integer
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ParamTuneLevel

Type: integer

Value: a valid integer

Default: 10

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Code Placement” on page 7-2
Selecting Persistence Level for Signals and Parameters

File packaging format

Method of file packaging for generated code files

Model Configuration Pane: Code Generation / Code Placement

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.

Settings

Modular (default) | Compact (with separate data file) | Compact

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 “Generate Subsystem Code as Separate Function and Files”.
- 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*.
- 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 Auto, and your model contains at least one of these blocks, separate files are generated for utility code in a shared location.
 - Model blocks
 - Simulink Function blocks
 - Function Caller blocks
 - Calls to Simulink Function blocks from Stateflow or MATLAB Function blocks
 - Stateflow graphical functions when you select the **Export Chart Level Functions** parameter
- 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 Auto, and your model contains at least one of these blocks, separate files are generated for utility code in a shared location.
 - Model blocks
 - Simulink Function blocks
 - Function Caller blocks
 - Calls to Simulink Function blocks from Stateflow or MATLAB Function blocks
 - Stateflow graphical functions when you select the **Export Chart Level Functions** parameter
- 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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ERTFilePackagingFormat

Type: character vector

Value: 'Modular' | 'CompactWithDataFile' | 'Compact'

Default: 'Modular'

Version History

Introduced in R2010a

See Also

Topics

“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”
“Generate Shared Utility Code”

Header files

Custom name for header files

Model Configuration Pane: Code Generation / Code Placement

Description

Specify customized name for generated header files.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

\$R\$E (default) | macro text

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. In files of other types, \$E does not generate text.
\$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 on page 15-46 parameter.

In the generated file name, adjacent tokens are separated by the underscore (_) character, even if a token does not generate text.

Custom naming is supported only for .h and .hpp files. When you have model hierarchy, custom naming is applicable to only the root model.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: ERTHeaderFileRootName

Type: character vector

Value: Valid combination of tokens and custom text

Default: \$R\$E

Version History

Introduced in R2018a

See Also

Topics

“Model Configuration Parameters: Code Generation Code Placement” on page 7-2

“Customize Generated File Names”

“Identifier Format Control”

Source files

Custom name for source files

Model Configuration Pane: Code Generation / Code Placement

Description

Specify customized name for generated source files.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

\$R\$E (default) | macro text

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. In files of other types, \$E does not generate text.
\$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 on page 15-46 parameter.

In the generated file name, adjacent tokens are separated by the underscore (_) character, even if a token does not generate text.

Custom naming is supported only for .c and .cpp files. When you have model hierarchy, custom naming is applicable to only the root model.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: ERTSourceFileRootName

Type: character vector

Value: Valid combination of tokens and custom text

Default: \$R\$E

Version History

Introduced in R2018a

See Also

Topics

“Model Configuration Parameters: Code Generation Code Placement” on page 7-2

“Customize Generated File Names”

“Identifier Format Control”

Data files

Custom name for data files

Model Configuration Pane: Code Generation / Code Placement

Description

Specify customized name for generated data files.

Dependencies

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 7-19 enables this parameter.

Settings

\$R_data (default) | macro text

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 on page 15-46 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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: ERTDataFileRootName

Type: character vector

Value: Valid combination of tokens and custom text

Default: \$R_data

Version History

Introduced in R2018a

See Also

Topics

“Model Configuration Parameters: Code Generation Code Placement” on page 7-2

“Customize Generated File Names”

“Identifier Format Control”

Rate Transition block code

Format for rate transition block code and data

Model Configuration Pane: Code Generation / Code Placement

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.

Dependencies

- This parameter requires an Embedded Coder license.
- Appears only for ERT-based targets.

Settings

Inline (default) | Function

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.

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 data transfer** 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.
-

Programmatic Use

Parameter: RateTransitionBlockCode

Value: 'Inline' | 'Function' |

Default: 'Inline'

Version History

Introduced in R2018a

See Also

Topics

“Time-Based Scheduling”

Code Generation Parameters: Code Style

Model Configuration Parameters: 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 for an ERT-based target require an Embedded Coder license.

You can change the code style, cast expressions, and indentation of your generated code to conform to certain coding standards.

Code style modifications have the following uses:

- Enhance the readability and traceability of code
- Convey information across files
- Enhance the efficiency of the generated code
- Allow memory manipulation through type casting

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Code Style** pane.

Parameter	Description
Parentheses level	Specify parenthesization style for generated code.
Preserve operand order in expression	Specify whether to preserve order of operands in expressions.
Preserve condition expression in if statement	Specify whether to preserve empty primary condition expressions in <code>if</code> statements.
Convert if-elseif-else patterns to switch-case statements	Specify whether to generate code for <code>if-elseif-else</code> decision logic as <code>switch-case</code> statements.
Preserve extern keyword in function declarations	Specify whether to include the <code>extern</code> keyword in function declarations in the generated code.
Preserve static keyword in function declarations	Specify whether to include the <code>static</code> keyword in function declarations in the generated code.
Suppress generation of default cases for switch statements if unreachable	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	Specify whether to replace multiplications by powers of two with signed bitwise shifts.
Allow right shifts on signed integers	Specify whether to allow signed right bitwise shifts in the generated C/C++ code.
Casting modes	Specify how the code generator casts data types for variables.
Static array container type	Specify the container type for arrays in the generated code. Choose either C-style array or <code>std::array</code> .

Parameter	Description
Indent style	Specify style for the placement of braces in generated code.
Indent size	Specify indent size for generated code.
Newline style	Specify the newline style for generated code.
Maximum line width	Specify the maximum line width for wrapping the generated code.

See Also

More About

- “Code Appearance”
- “Model Configuration Set Customization”

Code Generation: Code Style Tab Overview

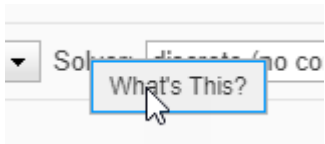
Control optimizations for readability in generated code.

Configuration

This tab appears only if you specify an ERT-based System target file.

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 Style” on page 8-2

Parentheses level

Parenthesization style for generated code

Model Configuration Pane: Code Generation / Code Style

Description

Specify parenthesization style for generated code.

Settings

Nominal (Optimize for readability) (default) | Minimum (Rely on C/C++ operators for precedence) | Standards (Parentheses for Standards Compliance) | Maximum (Specify precedence with parentheses)

Default: Nominal (Optimize for readability)

Minimum (Rely on C/C++ operators for precedence)

Inserts parentheses only where required by ANSI¹ C or C++, or to override default precedence.

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.

Standards (Parentheses for Standards Compliance)

Inserts parentheses in a way that requires both MISRA² compliance and better code readability. Code generated with this setting conforms to MISRA requirements.

Maximum (Specify precedence with parentheses)

Includes parentheses to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA requirements.

Examples

Control Level of Parentheses in Generated Code

Compare the parentheses in the generated code when you set **Parentheses level** to different values.

Here is generated code that generated using the default **Parentheses level** value Nominal (Optimize for readability):

```
Out = ((In2 * In1 - In1 > 1.0) && (In2 > 2.0));
```

Here is the same code generated with **Parentheses level** set to Minimum (Rely on C/C++ operators for precedence):

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.

```
Out = In2 * In1 - In1 > 1.0 && In2 > 2.0;
```

Here is the same code generated with **Parentheses level** set to Standards (Parentheses for Standards Compliance):

```
Out = (((In2 * In1) - In1 > 1.0) && (In2 > 2.0));
```

Here is the same code generated with **Parentheses level** set to Maximum (Specify precedence with parentheses):

```
Out = (((((In2 * In1) - In1) > 1.0) && (In2 > 2.0)));
```

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

Programmatic Use

Parameter: ParenthesesLevel

Type: character vector

Value: 'Minimum' | 'Nominal' | 'Standards' | 'Maximum'

Default: 'Nominal'

Version History

Introduced in R2006b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Control Parentheses in Generated Code”

Preserve operand order in expression

Preserve order of operands in expressions

Model Configuration Pane: Code Generation / Code Style

Description

Specify whether to preserve order of operands in expressions.

Settings

off (default) | on

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.

Note This setting applies to Embedded Coder only and might not affect order of operands inside Stateflow charts.

Off

Optimizes efficiency of code for nonoptimized compilers by reordering commutable operands to make expressions left-recursive.

Examples

Control Order of Operands in Generated Code Expressions

Compare the order of operands in expressions in the generated code when you set **Preserve operand order in expression** to different values.

Here is generated code that generated using the default **Preserve operand order in expression** value off:

$(B+C)*A$

Here is the same code generated with **Preserve operand order in expression** set to on:

$A*(B+C)$

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

Programmatic Use

Parameter: PreserveExpressionOrder

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Optimize Code by Reordering Commutable Operands”

Preserve condition expression in if statement

Preserve empty primary conditions in if statements

Model Configuration Pane: Code Generation / Code Style

Description

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

Settings

off (default) | on

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.

Off

Optimizes empty primary condition expressions in `if` statements by negating them.

Examples

Control Generation of Empty Primary Conditions in `if` Statements

Compare the `if` statements in the generated code when you set **Preserve condition expression in if statement** to different values.

Here is generated code that generated using the default **Preserve condition expression in if statement** value off:

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

Here is the same code generated with **Preserve condition expression in if statement** set to on:

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

Programmatic Use

Parameter: PreserveIfCondition

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Optimize Generated Code by Consolidating Redundant If-Else Statements”

Convert if-elseif-else patterns to switch-case statements

Generate code for if-elseif-else logic as switch-case statements

Model Configuration Pane: Code Generation / Code Style

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

Settings

on (default) | off

Default: on

On

Generate code for if-elseif-else decision logic as switch-case statements.

Off

Preserve if-elseif-else decision logic in generated code.

Examples

Control Generation of Switch-Case Statements for If-Elseif-Else Patterns

Compare the logic patterns in the generated code when you set **Convert if-elseif-else patterns to switch-case statements** to different values.

Here is generated code that generated using the default **Convert if-elseif-else patterns to switch-case statements** value off:

```
if (x == 1) {
    y = 1;
} else if (x == 2) {
    y = 2;
} else if (x == 3) {
    y = 3;
} else {
    y = 4;
}
```

Here is the same code generated with **Convert if-elseif-else patterns to switch-case statements** set to on:

```
switch (x) {  
    case 1:  
        y = 1; break;  
    case 2:  
        y = 2; break;  
    case 3:  
        y = 3; break;  
    default:  
        y = 4; break;  
}
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	No impact

Programmatic Use

Parameter: ConvertIfToSwitch

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2009b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Enhance Readability of Code for Flow Charts”

“Enhance Code Readability for MATLAB Function Blocks”

Preserve extern keyword in function declarations

Include extern keyword in function declarations

Model Configuration Pane: Code Generation / Code Style

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, such as model entry-point functions, global or scoped root-level Simulink functions, and referenced model functions. However, including the extern keyword in function declarations can enhance code readability.

Dependencies

When another model references this model, specify the same value that the parent model uses.

Settings

on (default) | off

Default: on

On

Include the extern keyword in function declarations in the generated code.

Off

Remove the extern keyword from function declarations in the generated code.

Examples

Control Generation of extern Keyword in Function Declarations

Compare the function declarations in the generated code when you set **Preserve extern keyword in function declarations** to different values.

Here is generated code that generated using the default **Preserve extern keyword in function declarations** value off:

```
/* Model entry point functions */
void CounterModel_initialize(void);
void CounterModel_step(void);
```

Here is the same code generated with **Preserve extern keyword in function declarations** set to on:

```
/* Model entry point functions */
extern void CounterModel_initialize(void);
extern void CounterModel_step(void);
```

The `extern` keyword explicitly indicates that the function has external linkage.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: `PreserveExternInFcnDecls`

Type: character vector

Value: `'on'` | `'off'`

Default: `'on'`

Version History

Introduced in R2009b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

Preserve static keyword in function declarations

Include `static` keyword in function declarations

Model Configuration Pane: Code Generation / Code Style

Description

Specify whether to include the `static` keyword in function declarations in the generated code.

Dependencies

- 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.

Settings

on (default) | off

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
- Simulink function inside a subsystem

This parameter does not impact these functions:

- Function in a model reference build
- Simulink function at root level

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.

Tips

- If your function cannot use the `static` keyword because it requires external access, consider modeling it as a Simulink function at the root level.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution, ROM)
Safety precaution	No impact

Programmatic Use

Parameter: PreserveStaticInFcnDecls

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2017b

See Also

MISRA C:2012 Rule 8.10 (Polyspace Bug Finder)

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Simulink Function Blocks and Code Generation”

Suppress generation of default cases for switch statements if unreachable

Whether to generate default cases for switch-case statements for Stateflow charts

Model Configuration Pane: Code Generation / Code Style

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”.

Settings

on (default) | off

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.

Examples

Control Generation of Unreachable Default Cases in Switch Statements

Compare the generated code when you set **Suppress generation of default cases for switch statements if unreachable** to different values.

Here is generated code that generated using the default **Suppress generation of default cases for switch statements if unreachable** value off. 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;
```

With **Suppress generation of default cases for switch statements if unreachable** set to on, the generated code does not contain the default case.

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	On
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

Programmatic Use

Parameter: SuppressUnreachableDefaultCases

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2011b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Suppress Generation of Default Cases for Unreachable Switch Statements”

Replace multiplications by powers of two with signed bitwise shifts

Replace multiplications by powers of two with signed bitwise shifts

Model Configuration Pane: Code Generation / Code Style

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.

Settings

on (default) | off

Default: on

On

Generate code that replaces multiplications by powers of two with signed bitwise shifts.

Off

Do not allow replacement of multiplications by powers of two with signed shifts. Clearing this option supports MISRA C compliance.

Examples

Control Generation of Multiplications by Powers of Two

Compare how the generated code implements multiplications by powers of two when you set **Replace multiplications by powers of two with signed bitwise shifts** to different values.

Here is generated code that generated using the default **Replace multiplications by powers of two with signed bitwise shifts** value on. Multiplications by 8 are left-shifted in the generated code:

```
Y.Out1 = (U.In1 << ((int8_T)3));
```

Here is the same code generated with **Replace multiplications by powers of two with signed bitwise shifts** set to off. Multiplications by 8 are not replaced by bitwise shifts:

```
Y.Out1 = U.In1 * ((int64_T)8);
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On

Application	Setting
Safety precaution	No impact

Programmatic Use

Parameter: EnableSignedLeftShifts

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2011b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Replace Multiplication by Powers of Two with Signed Bitwise Shifts”

Allow right shifts on signed integers

Allow right bitwise shifts on signed integers

Model Configuration Pane: Code Generation / Code Style

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.

Settings

on (default) | off

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);
```

Examples

Control Generation of Right Bitwise Shifts of Signed Integers

Compare how the generated code implements right bitwise shifts of signed integers when you set **Allow right shifts on signed integers** to different values.

Here is generated code that generated using the default **Allow right shifts on signed integers** value on. Right shifts appear in the generated code:

```
i >>= 3
```

Here is the same code generated with **Allow right shifts on signed integers** set to off. Right shifts are replaced by a function call:

```
i = asr_s32(i, 3U);
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

Programmatic Use

Parameter: EnableSignedRightShifts

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2015b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Generate Code Containing Right Shifts on Signed Integers”

Casting modes

Method of casting data types for variables

Model Configuration Pane: Code Generation / Code Style

Description

Specify how the code generator casts data types for variables.

Settings

Nominal (default) | Standards Compliant | Explicit

Default: Nominal

Nominal

Generate code that uses default C compiler data type casting. This setting removes unnecessary typecasts which might lead to warnings on compilers that expect explicit casts.

Standards Compliant

Generate code where data type casting complies with MISRA standards.

Setting **Casting modes** to **Standards Compliant** can replace bitwise XOR operations with relational operations in the generated code to satisfy the MISRA C:12 Rule 10.1 when the operands are signed types.

Explicit

Generate code that casts data type values explicitly.

Examples

Control Data Type Casting for Variables

Compare the data type casting in the generated code when you set **Casting modes** to different values.

Here is generated code that generated using the default **Casting modes** value Nominal:

```
void EmbeddedCoderIntro_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;
    }
}
```

Here is the same code generated with **Casting modes** set to **Standards Compliant**:

```
void EmbeddedCoderIntro_step(void)
{
    boolean_T rtb_equal_to_count;
    rtDWork.X = (uint8_T)((uint32_T)rtDWork.X + 1U);
    rtb_equal_to_count = ((int32_T)rtDWork.X != 16);
}
```

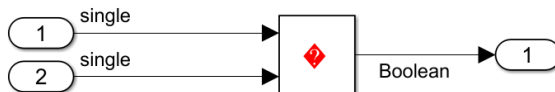
```

if (rtb_equal_to_count && ((uint32_T)rtPrevZCSigState.Amplifier_Trig_ZCE !=
    POS_ZCSIG)) {
    rtY.Output = rtU.Input << 1U;
}

```

Note The expression `rtY.Output = rtU.Input << 1U` is not compliant with MISRA C:12 Rule 10.1 because the model configuration parameter **Replace multiplications by powers of two with signed bitwise shift** is selected. For more information, see Replace multiplications by powers of two with signed bitwise shifts.

Depending on the setting, the configuration parameter **Casting modes** can replace bitwise XOR operations with relational operations in the generated code to satisfy the MISRA C:12 Rule 10.1 when the operands are signed types. For example, generate code from the following model with the **Casting modes** set to Nominal and Standard compliant respectively.



```

// Model step function (casting mode set to Nominal)
void step(void)
{rtY.Out3 = (boolean_T)((int32_T)(rtU.In1 != 0.0F) ^ (int32_T)(rtU.Inport1 !=
    0.0F));
}
// Model step function (Casting modes set to Standard Compliant)
void step(void)
{ rtY.Out3 = ((rtU.In1 != 0.0F) != (rtU.Inport1 != 0.0F));
}

```

Here, the parameters `rtU.In1` and `rtU.Inport1` are single signed types. Performing a bitwise XOR(^) operation on these operands violates the MISRA C:12 Rule 10.1. To prevent this violation, the code generator replaces the bitwise XOR(^) operation with an inequality(!=) in the generated code when **Casting modes** is set to Standard compliant.

Here is the same code generated with **Casting modes** set to Explicit:

```

void EmbeddedCoderIntro_step(void)
{
    boolean_T rtb_equal_to_count;
    rtDWork.X = (uint8_T)((uint32_T)(int32_T)rtDWork.X + 1U);
    rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);
    if (((int32_T)rtb_equal_to_count) && ((int32_T)
        rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG)) {
        rtY.Output = rtU.Input << 1;
    }
}

```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: CastingMode

Type: character vector

Value: 'Nominal' | 'Standards' | 'Explicit'

Default: 'Nominal'

Version History

Introduced in R2014b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“MISRA C”

Static array container type

Container type for static arrays

Model Configuration Pane: Code Generation / Code Style

Description

Specify a container type for static arrays in the generated code. Choose either `C-style array` or `std::array`.

Dependencies

To enable this parameter, set **Language** to C++.

Settings

`C-style array` (default) | `std::array`

Default: C-style array

C-style array

The code generator generates array containers by using C-style arrays.

std::array

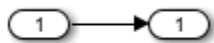
`std::array` is a template class that represents fixed-size arrays. If you choose this option, the code generator generates array containers by using `std::array`.

If the generated code contains multidimensional arrays, the code generator does not honor the `std::array` container type for arrays of more than one dimension.

Examples

Control Container Type for Static Arrays

Consider this model, `myModel`:



The model passes a 1-dimensional signal of length 5 from an Inport block to an Outport block. If you set the **Static array container type** model configuration parameter to `C-style array`, the code generator produces this code in `myModel.h`:

```
// External inputs (root inport signals with default storage)
struct ExtU_myModel_T {
    real_T Inport[5];           // '<Root>/Inport'
};

// External outputs (root outports fed by signals with default storage)
struct ExtY_myModel_T {
```

```
    real_T Output[5];                // '<Root>/Output'
};
```

If you set the **Static array container type** model configuration parameter to `std::array`, the code generator produces this code instead:

```
// External inputs (root inport signals with default storage)
struct ExtU_myModel_T {
    std::array<real_T, 5> Inport;      // '<Root>/Inport'
};

// External outputs (root outputs fed by signals with default storage)
struct ExtY_myModel_T {
    std::array<real_T, 5> Output;      // '<Root>/Output'
};
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: ArrayContainerType

Type: character vector

Value: 'C-style array' | 'std::array'

Default: 'C-style array'

Limitations

- This parameter does not affect static arrays in function interfaces. If a generated function takes a fixed-size array as an argument or returns a fixed-size array, the code generator generates that array as a C-style array regardless of the setting of this parameter.

Version History

Introduced in R2020a

R2024b: Renamed to Static array container type

To differentiate from the new **Dynamic array container type**, the **Array container type** model configuration parameter is now named **Static array container type**. The programmatic name of the parameter is still `ArrayContainerType`. There are no plans to change the programmatic name.

See Also

Dynamic array container type

Topics

“Model Configuration Parameters: Code Style” on page 8-2

Indent style

Style for placement of braces

Model Configuration Pane: Code Generation / Code Style

Description

Specify style for the placement of braces in generated code.

Settings

K&R (default) | Allman

Default: K&R

K&R

For blocks within a function, an opening brace is on the same line as its control statement.

Allman

For blocks within a function, an opening brace is on its own line at the same level of indentation as its control statement.

Examples

Control Indent Style in Generated Code

Compare the array container types in the generated code when you set **Indent style** to different values.

Here is generated code that generated using the default **Indent style** value K&R:

```
void rt_OneStep(void)
{
    static boolean_T OverrunFlag = false;

    /* Disable interrupts here */

    /* Check for overrun */
    if (OverrunFlag) {
        rtmSetErrorStatus(rtM, "Overrun");
        return;
    }

    OverrunFlag = true;

    /* Save FPU context here (if necessary) */
    /* Re-enable timer or interrupt here */
    /* Set model inputs here */

    /* Step the model */
    CounterModel_step();
```

```
/* Get model outputs here */

/* Indicate task complete */
OverrunFlag = false;

/* Disable interrupts here */
/* Restore FPU context here (if necessary) */
/* Enable interrupts here */
}
```

Here is the same code generated with **Indent style** set to Allman:

```
void rt_OneStep(void)
{
    static boolean_T OverrunFlag = false;

    /* Disable interrupts here */

    /* Check for overrun */
    if (OverrunFlag)
    {
        rtmSetErrorStatus(rtM, "Overrun");
        return;
    }

    OverrunFlag = true;

    /* Save FPU context here (if necessary) */
    /* Re-enable timer or interrupt here */
    /* Set model inputs here */

    /* Step the model */
    CounterModel_step();

    /* Get model outputs here */

    /* Indicate task complete */
    OverrunFlag = false;

    /* Disable interrupts here */
    /* Restore FPU context here (if necessary) */
    /* Enable interrupts here */
}
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: IndentStyle

Type: character vector

Value: 'K&R' | 'Allman'

Default: 'K&R'

Version History

Introduced in R2013b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Replace and Rename Simulink Coder Data Types to Conform to Coding Standards”

Indent size

Size for indents in generated code

Model Configuration Pane: Code Generation / Code Style

Description

Specify indent size for generated code.

Settings

2 | 3 | 4 | 5 | 6 | 7 | 8

Default: 2

Specify an integer value that indicates the number of characters per indent level. Possible values range from 2-8 characters.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: IndentSize

Type: integer

Value: integer from 2-8

Default: 2

Version History

Introduced in R2013b

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Replace and Rename Simulink Coder Data Types to Conform to Coding Standards”

Newline style

Newline character in generated code

Model Configuration Pane: Code Generation / Code Style

Description

Specify the newline character in the generated code.

Settings

Default (default) | LF (Line Feed) | CR+LF (Carriage Return + Line Feed)

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: NewlineStyle

Type: character vector

Value: 'Default'|'LF'|'CR+LF'

Default: 'Default'

Version History

Introduced in R2018a

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Replace and Rename Simulink Coder Data Types to Conform to Coding Standards”

Maximum line width

Maximum line width for wrapping generated code

Model Configuration Pane: Code Generation / Code Style

Description

Specify the maximum line width for wrapping generated code.

Settings

80 (default) | value from 50-1000

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

Examples

Wrap Generated Code According to Maximum Line Width

Compare how the generated code wraps when you set **Maximum line width** to 80 and 120.

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 */
```

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: MaxLineWidth

Type: integer

Value: integer from 50–1000

Default: 80

Version History

Introduced in R2019a

See Also

Topics

“Model Configuration Parameters: Code Style” on page 8-2

“Replace and Rename Simulink Coder Data Types to Conform to Coding Standards”

Dynamic array container type

Container type for dynamic arrays

Model Configuration Pane: Code Generation / Code Style

Description

Specify a container type for 1-dimensional dynamic arrays in the generated code. Choose either `coder::array` or `std::vector`.

This parameter affects only 1-dimensional dynamic arrays. For higher-dimensional dynamic arrays, the code generator generates `coder::array` containers regardless of the value of this parameter.

Dependencies

- To enable this parameter, set the **Language** model configuration parameter to C++.

Settings

`std::vector` (default) | `coder::array`

Default: `std::vector`

`std::vector`

The code generator generates `std::vector` containers to represent 1-dimensional dynamic arrays.

`coder::array`

The code generator generates `coder::array` containers to represent 1-dimensional dynamic arrays. If you open a model that was created before R2024b, the model defaults to this setting.

Examples

Control Container Type for Dynamic Arrays

Consider this model, `myModel`:



The model passes an unbounded variable-size 1-dimensional signal from an Inport block to an Outport block. If you set the **Dynamic array container type** model configuration parameter to `std::vector`, the code generator produces this code in `myModel.h`:

```
// External inputs (root inport signals with default storage)
struct ExtU_myModel_T {
    std::vector<real_T> Input;          // '<Root>/Input'
};
```

```
// External outputs (root outputs fed by signals with default storage)
struct ExtY_myModel_T {
    std::vector<real_T> Out1;          // '<Root>/Out1'
};
```

If you set the **Dynamic array container type** model configuration parameter to `coder::array`, the code generator produces this code instead:

```
// External inputs (root inport signals with default storage)
struct ExtU_myModel_T {
    coder::array<real_T,1> Input;      // '<Root>/Input'
};

// External outputs (root outputs fed by signals with default storage)
struct ExtY_myModel_T {
    coder::array<real_T,1> Out1;       // '<Root>/Out1'
};
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: DynamicArrayContainerType
Type: character vector
Values: 'std::vector' | 'coder::array'
Default: 'std::vector'

Version History

Introduced in R2024b

See Also

Static array container type

Topics

“Model Configuration Parameters: Code Style” on page 8-2

Code Generation Parameters: Code Generation

- “Model Configuration Parameters: Code Generation” on page 9-2
- “Set Objectives” on page 9-12
- “Set Objectives — Code Generation Advisor Dialog Box” on page 9-13

Model Configuration Parameters: Code Generation

The **Code Generation** category includes parameters for defining the code generation process including target selection. It also includes parameters for inserting comments and pragmas into the generated code for data and functions. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license. Generating CUDA® C++ code for NVIDIA® GPUs requires a GPU Coder license.

These configuration parameters appear in the **Configuration Parameters > Code Generation** general category.

Parameter	Description
System target file	Specify which target file configuration will be used.
"Browse"	Browse file configuration options.
Shared coder dictionary	Shared coder dictionary that contains code interface configuration.
Language	Specify C or C++ code generation.
Language standard	Specify the language standard for your execution environment.
Generate GPU code	Use GPU Coder for CUDA code generation. This parameter requires a GPU Coder license.
"Description"	A description of the target file.
Generate code only	Specify code generation versus an executable build.
Package code and artifacts	Specify whether to automatically package generated code and artifacts for relocation.
Zip file name	Specify the name of the .zip file in which to package generated code and artifacts for relocation.
Compiler optimization level	Control compiler optimizations for building generated code.
Custom compiler optimization flags	Specify custom compiler optimization flags.
Toolchain	Specify the toolchain to use when building an executable or library.
Build configuration	Specify compiler optimization or debug settings for toolchain.
Toolchain details	Display or customize build configuration settings.
Generate makefile	Enable generation of a makefile based on a template makefile.
Make command	Specify a make command and optionally append makefile options.

Parameter	Description
Template makefile	Specify the template makefile from which to generate the makefile.
Select objective	Select a code generation objective to use with the Code Generation Advisor.
Prioritized objectives	List of prioritized code generation objectives.
“Set Objectives” on page 9-12	Open Configuration Set Objectives dialog box.
“Set Objectives — Code Generation Advisor Dialog Box” on page 9-13	Select and prioritize code generation objectives.
Check model before generating code	Choose whether to run Code Generation Advisor checks before generating code.
“Check Model”	Check whether the model meets code generation objectives.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
Built-in FFTW library callback	Use the built-in callback for FFTW library usage in code generated for FFT functions in MATLAB code.
Custom FFT library callback	Specify a callback class for FFTW library calls in code generated for FFT functions in MATLAB code.
Custom BLAS library callback	Specify BLAS library callback class for BLAS calls in code generated from MATLAB code.
Custom LAPACK library callback	Specify LAPACK library callback class for LAPACK calls in code generated from MATLAB code.
Verbose build	Display code generation progress.
Retain .rtw file	Specify <i>model.rtw</i> file retention.
Profile TLC	Profile the execution time of TLC files.
Enable TLC assertion	Produce the TLC stack trace.
Start TLC coverage when generating code	Generate the TLC execution report.
Start TLC debugger when generating code	Specify use of the TLC debugger
Show Custom Hardware App in Simulink Toolstrip	Read-only internal parameter for Simulink toolstrip.
Show Embedded Hardware App in Simulink Toolstrip	Read-only internal parameter for Simulink toolstrip.
“Package” on page 17-3	Specify a package that contains memory sections you want to apply to model-level functions and internal data.
“Refresh package list” on page 17-5	Add user-defined packages that are on the search path to list of packages.

Parameter	Description
"Initialize/Terminate" on page 17-6	Specify whether to apply a memory section to Initialize/Start and Terminate functions.
"Execution" on page 17-7	Specify whether to apply a memory section to execution functions.
"Shared utility" on page 17-9	Specify whether to apply memory sections to shared utility functions.
"Constants" on page 17-11	Specify whether to apply a memory section to constants.
"Inputs/Outputs" on page 17-13	Specify whether to apply a memory section to root input and output.
"Internal data" on page 17-15	Specify whether to apply a memory section to internal data.
"Parameters" on page 17-17	Specify whether to apply a memory section to parameters.
"Validation results" on page 17-19	Display the results of memory section validation.

The following parameters under **Advanced parameters** are infrequently used and have no other documentation.

Parameter	Description
PostCodeGenCommand <i>character vector</i> - ''	Add the specified post code generation command to the model build process.
TLCOptions <i>character vector</i> - ''	Specify additional TLC command-line options. When you use Embedded Coder for a component model that is configured with a service interface, you cannot specify - aInlineSetEventsForThisBaseRateFcn=TLC_FALSE, - aSuppressMultiTaskScheduler=TLC_FALSE, - aRateBasedStepFcn=TLCFALSE, - (wrapper function)

The following parameters are for MathWorks use only.

Parameter	Description
Comment	For MathWorks use only.
PreserveName	For MathWorks use only.
PreserveNameWithParent	For MathWorks use only.
SignalNamingFcn	For MathWorks use only.

Parameter	Description
TargetTypeEmulationWarnSuppressLevel int - 0	For MathWorks use only. When greater than or equal to 2, suppress warning messages that the code generator displays when emulating integer sizes in rapid prototyping environments.

The Configuration Parameters dialog box also includes other code generation parameters:

- “Model Configuration Parameters: Code Generation Optimization”
- “Model Configuration Parameters: Code Generation Report”
- “Model Configuration Parameters: Comments”
- “Model Configuration Parameters: Code Generation Identifiers”
- “Model Configuration Parameters: Code Generation Custom Code”
- “Model Configuration Parameters: Code Generation Interface”

See Also

More About

- “Model Configuration Set Customization”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Shared coder dictionary

Embedded Coder Dictionary that contains code interface configuration for code generation

Model Configuration Pane: Code Generation

Description

The **Shared coder dictionary** parameter specifies the Embedded Coder Dictionary that contains the code interface configuration for code generation. The dictionary must be an `.sldd` file that is on the MATLAB path.

Dependencies

To enable this parameter:

- Install Embedded Coder.
- Set **System target file** to an ERT-based system target file.

Settings

' ' (default) | string

Specify the coder dictionary by using one of these options:

- Enter the name of your `.sldd` file in this field.
- Use the Embedded Coder Dictionary browser. Click **Set up** and select an `.sldd` file.
- Create an Embedded Coder Dictionary. Click **Set up** and create an `.sldd` file that contains an Embedded Coder Dictionary.

Recommended Settings

Application	Setting
Debugging	No recommendation
Traceability	No recommendation
Efficiency	No recommendation
Safety precaution	No recommendation

Programmatic Use

Parameter: EmbeddedCoderDictionary

Type: character vector

Value: name of a valid `.SLDD` file that contains an Embedded Coder Dictionary

Default: ' '

Version History

Introduced in R2022b

See Also

Topics

“Model Configuration Parameters: Code Generation” on page 9-2
Embedded Coder Dictionary

Prioritized objectives

Prioritized list of Code Generation Advisor objectives to apply for reviewing model configuration parameter settings

Model Configuration Pane: Code Generation

Description

List of objectives that you specify in order of priority for reviewing model configuration parameter settings by using the Code Generation Advisor. To select and prioritize objectives, click **Set Objectives** and make changes in the Set Objectives - Code Generation Advisor dialog box.

Dependencies

- This parameter appears only when **System target file** is set to an ERT-based system target file. For GRT-based system target file configurations, select an objective by using the parameter Select objective.
- To generate code, this parameter requires Embedded Coder.

Settings

Unspecified (default) | Execution efficiency | ROM efficiency | RAM efficiency | Traceability | Safety precaution | Debugging | MISRA C:2023 guidelines | MISRA CPP:2023 guidelines | Polyspace | ...

Default: Unspecified

Unspecified

No objective specified. Do not optimize code generation settings by using the Code Generation Advisor.

Execution efficiency

Configure code generation parameter settings to achieve fast execution time.

ROM efficiency

Configure code generation parameter settings to reduce ROM usage.

RAM efficiency

Configure code generation parameter settings to reduce RAM usage.

Traceability

Configure code generation parameter settings to provide mapping between model elements and code.

Safety precaution

Configure code generation parameter settings to increase clarity, determinism, robustness, and verifiability of the code.

Debugging

Configure code generation parameter settings to debug the code generation build process.

MISRA C:2023 guidelines

Configure code generation parameter settings to increase compliance with MISRA C:2023 guidelines.

MISRA CPP:2023 guidelines

Configure code generation parameter settings to increase compliance with MISRA C++:2023 guidelines.

Polyspace

Configure code generation parameter settings to prepare the code for Polyspace® analysis.

Note If you select the MISRA C:2023 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA C:2023 configuration setting recommendations.
 - For blocks that are not supported or recommended for MISRA C:2023 compliant code generation.
-

Recommended Settings

Application	Setting
Debugging	Debugging
Traceability	Traceability
Efficiency	Execution efficiency (execution), ROM efficiency (ROM), RAM efficiency (RAM)
Safety precaution	Safety precaution

Programmatic Use

Parameter: ObjectivePriorities

Type: cell array of character vectors or string array

Value: {' '} | {'Execution efficiency'} | {'ROM efficiency'} | {'RAM efficiency'} | {'Traceability'} | {'Safety precaution'} | {'Debugging'} | {'MISRA C:2023 guidelines'} | {'MISRA CPP:2023 guidelines'} | {'Polyspace'} | cell array of objective names

Default: {' '}

Version History

Introduced in R2009a

See Also

Topics

“Model Configuration Parameters: Code Generation” on page 9-2

“Set Objectives — Code Generation Advisor Dialog Box” on page 9-13

“Configure Model for Code Generation Objectives by Using Code Generation Advisor”

“Application Objectives Using Code Generation Advisor”

Generate Halide code

Generate Halide code for faster array computation

Model Configuration Pane: Code Generation

Description

The **Generate Halide code** parameter instructs the code generator to generate Halide code. If there are no opportunities for Halide code generation, Embedded Coder will generate plain C/C++ code.

Dependencies

To enable this parameter:

- Install Embedded Coder.
- Set **System target file** to an ERT-based system target file.
- Set **Array layout** to Column-major.
- Disable **Generate GPU code**.
- Set **Code replacement libraries** to None.
- If the **Language** is set to C, then set **Hardware board** to None.
- If the **Language** is set to C++, then set **Hardware board** to one of these settings:
 - Android Device
 - Android Device (64bit)
 - Raspberry Pi
 - Raspberry Pi (64bit)
 - Raspberry Pi - Robot Operating System (ROS)
 - None

Settings

off (default) | on

On

Generates Halide code.

Off

Does not generate Halide code.

Recommended Settings

Application	Setting
Debugging	No recommendation
Traceability	No recommendation

Application	Setting
Efficiency	Yes
Safety precaution	No recommendation

Programmatic Use

Parameter: HalideCodeGeneration

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2023b

See Also

Topics

“Speed Up Generated Code Execution with Halide Code”

“Enhance Generated Code Performance Using Halide from a MATLAB Function Block”

“Model Configuration Parameters: Code Generation”

External Websites

www.halide-lang.org

Set Objectives

Description

Open Set Objectives - Code Generation Advisor dialog box.

Category: Code Generation

Dependency

This button appears only for ERT-based targets.

See Also

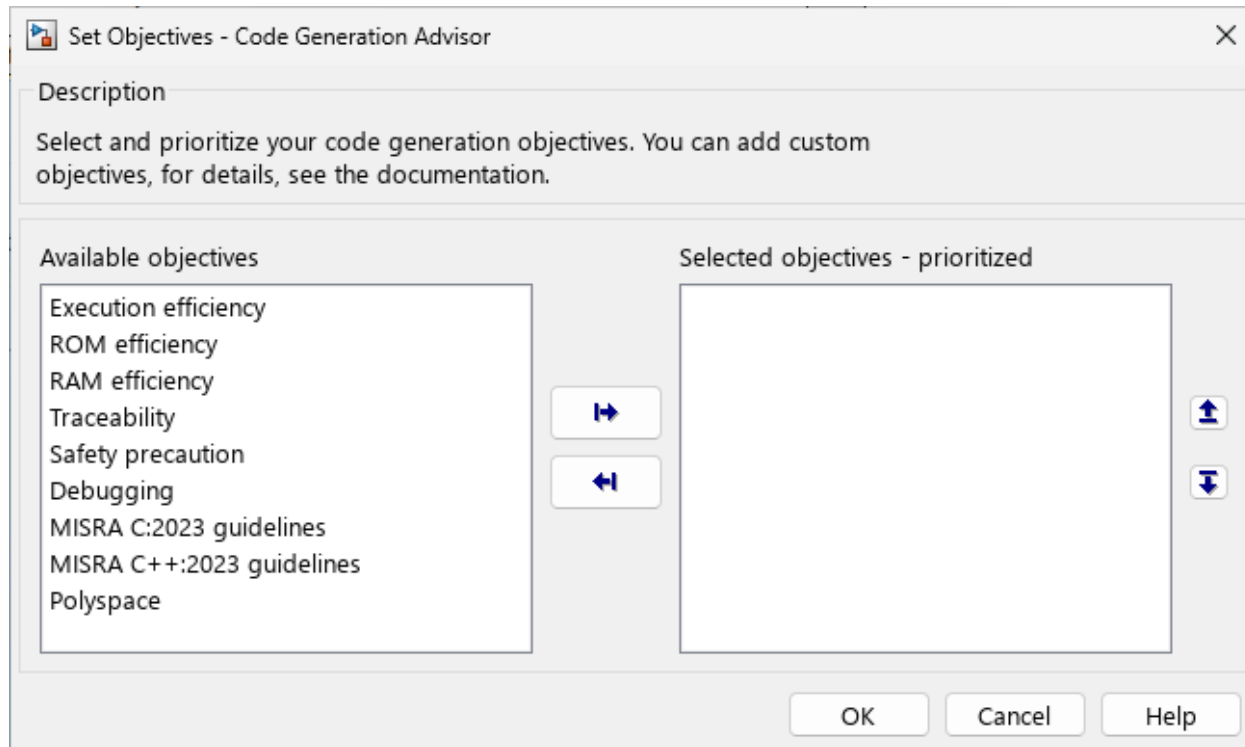
Related Examples

- “Model Configuration Parameters: Code Generation” on page 9-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor”
- “Application Objectives Using Code Generation Advisor”

Set Objectives — Code Generation Advisor Dialog Box

Description

Select and prioritize code generation objectives to use with the Code Generation Advisor.



Category: Code Generation

Settings

- 1 From the **Available objectives** list, select objectives.
- 2 Click the select button (arrow pointing right) to move the objectives that you selected into the **Selected objectives - prioritized** list.
- 3 Click the higher priority (up arrow) and lower priority (down arrow) buttons to prioritize the objectives.

Objectives

List of available objectives.

Execution efficiency — Configure code generation settings to achieve fast execution time.

ROM efficiency — Configure code generation settings to reduce ROM usage.

RAM efficiency — Configure code generation settings to reduce RAM usage.

Traceability — Configure code generation settings to provide mapping between model elements and code.

Safety precaution — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.

Debugging — Configure code generation settings to debug the code generation build process.

MISRA C:2023 guidelines — Configure code generation settings to increase compliance with MISRA C:2023 guidelines.

MISRA C++:2023 guidelines — Configure code generation settings to increase compliance with MISRA C++:2023 guidelines.

Polyspace — Configure code generation settings to prepare the code for Polyspace analysis.

Note If you select the MISRA C:2023 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA C:2023 configuration setting recommendations.
 - For blocks that are not supported or recommended for MISRA C:2023 compliant code generation.
-

Priorities

After you select objectives from the **Available objectives** parameter, organize the objectives in the **Selected objectives - prioritized** parameter with the highest priority objective at the top.

Dependency

This dialog box appears only for ERT-based targets.

Command-Line Information

Parameter: 'ObjectivePriorities'

Type: cell array of character vectors or string array; combination of the available values

Value: {' '} | {'Execution efficiency'} | {'ROM efficiency'} | {'RAM efficiency'} | {'Traceability'} | {'Safety precaution'} | {'Debugging'} | {'MISRA C:2023 guidelines'} | {'MISRA CPP:2023 guidelines'} | {'Polyspace'}

Default: {' '}

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 9-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor”
- “Application Objectives Using Code Generation Advisor”

Code Generation Parameters: Optimization

Model Configuration Parameters: Code Generation Optimization

The **Code Generation > Optimization** category includes parameters for improving the simulation speed of your models and improving the performance of the generated code. Model configuration parameters to improve the generated code require Simulink Coder or Embedded Coder.

Parameter	Description
Default parameter behavior	Transform numeric block parameters into constant inlined values in the generated code.
Leverage target hardware instruction set extensions	Select instruction sets to generate SIMD (single instruction, multiple data) code for target hardware.
Optimize reductions	Generate single instruction, multiple data (SIMD) code for reduction operation loops.
Pass reusable subsystem outputs as	Specify how a reusable subsystem passes outputs.
Remove root level I/O zero initialization	Specify whether to generate initialization code for root-level inports and outports set to zero.
Remove internal data zero initialization	Specify whether to generate initialization code for internal work structures, such as block states and block outputs, to zero.
Level	Choose the optimization level that you want to apply to the generated code.
Priority	Optimize the generated code for increased execution efficiency, decreased RAM consumption, or a balance between the two.
Specify custom optimizations	Instead of applying an optimization level, select this parameter to select the optimization parameters in the Details section.
Use memcpy for vector assignment	Optimize code generated for vector assignment by replacing <code>for</code> loops with <code>memcpy</code> .
Memcpy threshold (bytes)	Specify the minimum array size in bytes for which <code>memcpy</code> and <code>memset</code> function calls should replace <code>for</code> loops for vector assignments in the generated code.
Enable local block outputs	Specify whether block signals are declared locally or globally.
Reuse local block outputs	Specify whether Simulink Coder software reuses signal memory.
Eliminate superfluous local variables (Expression folding)	Collapse block computations into single expressions.
Reuse global block outputs	Reuse global memory for block outputs.

Parameter	Description
Reuse output buffers of Model blocks	Reuse referenced model buffers if possible.
Unify buffer reuse candidates	Analyze buffer reuse candidates at model-level to implement the optimal reuse in the generated code.
Perform in-place updates for Assignment and Bus Assignment blocks	Reuse the input and output variables of Bus Assignment and Assignment blocks if possible.
Reuse buffers for Data Store Read and Data Store Write blocks	Remove temporary buffers for Data Store Read and Data Store Write blocks. Use the Data Store Memory block directly if possible.
Simplify array indexing	Replace multiply operations in array indices when accessing arrays in a loop.
Pack Boolean data into bitfields	Specify whether Boolean signals are stored as one-bit bitfields or as a Boolean data type.
Bitfield declarator type specifier	Specify the bitfield type when selecting configuration parameter Pack Boolean data into bitfields.
Reuse buffers of different sizes and dimensions	Reduce memory consumption by reusing buffers to store data of different sizes and dimensions.
Optimize global data access	Select global variable optimization.
Optimize block operation order in generated code	Reorder block operations in the generated code for improved code execution speed.
Use bitsets for storing state configuration	Use bitsets to reduce the amount of memory required to store state configuration variables.
Use bitsets for storing Boolean data	Use bitsets to reduce the amount of memory required to store Boolean data.
Maximum stack size (bytes)	Specify the maximum stack size in bytes for your model.
Loop unrolling threshold	Specify the minimum signal or parameter width for which a <code>for</code> loop is generated.
Optimize using the specified minimum and maximum values	Optimize generated code using the specified minimum and maximum values for signals and parameters in the model.
Maximum number of arguments for subsystem outputs	Set maximum number of subsystem outputs to pass individually.
Inline invariant signals	Transform symbolic names of invariant signals into constant values.
Remove code from floating-point to integer conversions with saturation that maps NaN to zero	Remove code that handles floating-point to integer conversion results for NaN values.
Use memset to initialize floats and doubles to 0.0	Specify whether to generate code that explicitly initializes floating-point data to 0.0.
Remove code from floating-point to integer conversions that wraps out-of-range values	Remove wrapping code that handles out-of-range floating-point to integer conversion results.

Parameter	Description
Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values	Remove wrapping code of tunable parameters.
Remove code that protects against integer division arithmetic exceptions	Specify whether to generate code that guards against division by zero and INT_MIN/-1 operations for integers and fixed-point data.
Buffer for reusable subsystems	Improve reuse by inserting buffers at reusable subsystem boundaries.
Disable incompatible optimizations	Specify whether to disable optimizations that are incompatible with Simulink Code Inspector.
Base storage type for automatically created enumerations	Set the storage type and size for enumerations created with active state output.
Use signal labels to guide buffer reuse	For signals with the same label, the code generator attempts to use the same signal memory.
Generate parallel for-loops	Specify whether for-loops in the generated code shall be implement in parallel for Matlab Function, Matlab System or a For Each block.
Automatically schedule for-loops	Specify whether for-loop nests in the generated code are automatically scheduled for Neighborhood Processing blocks.
Use C restrict qualifier	Specify whether to apply the C restrict keyword to function arguments.
Signal storage reuse	Specify reuse of memory buffers allocated to store block input and output signals thereby reducing the memory requirement of real-time program.
Operator to represent Bitwise and Logical Operator blocks	Specify whether the generated code contains bitwise or logical operators or both.
Generate Halide code	Generate Halide code for faster array computation.
Function inlining between user functions	Control inlining for calls from user-written functions to other user-written functions.
Function inlining between MathWorks functions	Control inlining for calls from MathWorks functions to other MathWorks functions.
Function inlining between user and MathWorks functions	Control inlining for calls from user-written functions to MathWorks functions and from MathWorks functions to user-written functions.

See Also

Related Examples

- “Code Efficiency”

Pass reusable subsystem outputs as

Control how a reusable subsystem passes outputs

Model Configuration Pane: Code Generation / Optimization

Description

The **Pass reusable subsystem outputs as** parameter specifies how a reusable subsystem passes outputs.

Settings

Individual arguments (default) | Structure reference

Individual arguments

Passes each reusable subsystem output argument as an address of a local, instead of as a pointer to an area of global memory containing the output arguments. This option reduces global memory usage and eliminates copying local variables back to global block I/O structures. When the signals are allocated as local variables, there may be an increase in stack size. If the stack size increases beyond a level that you want, use the default setting. By default, the maximum number of output arguments passed individually is 12. To increase the number of arguments, increase the value of the **Maximum number of arguments for subsystem outputs** parameter.

Structure reference

Passes reusable subsystem outputs as a pointer to a structure stored in global memory.

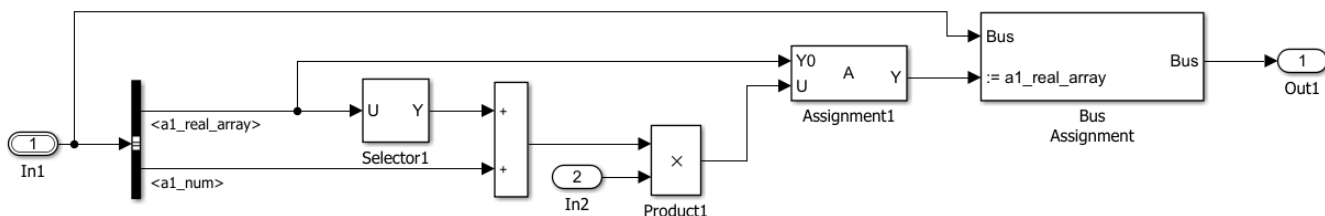
Note The default option is used for reusable subsystems that have signals with variable dimensions.

Examples

Reduce Data Copies for Bus Assignment Blocks in Generated Code

For models containing a Bus Assignment block, if possible, the code generator uses the same variable for the block input and output. Reusing these variables reduces data copies, conserves RAM consumption and increases code execution speed.

For example, in `bus_assignoptim`, a bus signal containing six elements feeds into a Bus Assignment block and a Bus Selector block. The Bus Assignment block assigns new values to the bus element `a1_real_array`. This bus signal feeds into `Out1`.



Generate code without the optimization by setting the **Perform in-place updates for Assignment and Bus Assignment blocks** parameter to off. Without the optimization the `bus_assignoptim_step` function contains this code:

```
void bus_assignoptim_step(void)
{
    real_T rtb_Assignment[36];
    int32_T i;

    /* Assignment: '<Root>/Assignment' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Product: '<Root>/Product'
     * Selector: '<Root>/Selector'
     * Sum: '<Root>/Sum1'
     */
    for (i = 0; i < 36; i++) {
        rtb_Assignment[i] = bus_assignoptim_U.In1.al_real_array[i];
    }

    for (i = 0; i < 2; i++) {
        rtb_Assignment[(int32_T)(i + 22)] = (bus_assignoptim_U.In1.al_real_array
            [(int32_T)(i + 22)] + bus_assignoptim_U.In1.al_num) *
            bus_assignoptim_U.In2;
    }

    /* End of Assignment: '<Root>/Assignment' */

    /* Output: '<Root>/Out' incorporates:
     * BusAssignment: '<Root>/Bus Assignment'
     * Inport: '<Root>/In1'
     */
    bus_assignoptim_Y.Out = bus_assignoptim_U.In1;

    /* BusAssignment: '<Root>/Bus Assignment' incorporates:
     * Output: '<Root>/Out'
     */
    for (i = 0; i < 36; i++) {
        bus_assignoptim_Y.Out.al_real_array[i] = rtb_Assignment[i];
    }
}
```

The generated code contains the temporary array `rtb_Assignment` for holding data before this data is assigned to `bus_assignoptim_Y.Out.al_real_array`.

Generate code with the optimization by setting the **Perform in-place updates for Assignment and Bus Assignment blocks** parameter to on. With the optimization, the `bus_assignoptim_step` function contains this code:

```
/* Model step function */
void bus_assignoptim_step(void)
{
    int32_T i;

    /* Output: '<Root>/Out' incorporates:
     * Inport: '<Root>/In1'
     * SignalConversion: '<Root>/TmpBusAssignmentBufferAtBus AssignmentInport1'
     */
}
```

```

bus_assignoptim_Y.Out = bus_assignoptim_U.In1;

/* Assignment: '<Root>/Assignment' incorporates:
 * Inport: '<Root>/In1'
 * Inport: '<Root>/In2'
 * Outport: '<Root>/Out'
 * Product: '<Root>/Product'
 * Selector: '<Root>/Selector'
 * Sum: '<Root>/Sum1'
 */
for (i = 0; i < 36; i++) {
    bus_assignoptim_Y.Out.al_real_array[i] =
        bus_assignoptim_U.In1.al_real_array[i];
}

for (i = 0; i < 2; i++) {
    bus_assignoptim_Y.Out.al_real_array[(int32_T)(i + 22)] =
        (bus_assignoptim_U.In1.al_real_array[(int32_T)(i + 22)] +
         bus_assignoptim_U.In1.al_num) * bus_assignoptim_U.In2;
}

/* End of Assignment: '<Root>/Assignment' */
}

```

The generated code does not contain the temporary array `rtb_Assignment1` for holding data. The generated code directly assigns this data to `bus_assignoptim_Y.Out.al_real_array`.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Individual arguments (execution, RAM), Structure reference (ROM)
Safety precaution	No impact

Programmatic Use

Parameter: `PassReuseOutputArgsAs`

Type: character vector

Value: 'Structure reference' | 'Individual arguments'

Default: 'Individual arguments'

Version History

Introduced in R2009a

See Also

"Model Configuration Parameters: Code Generation Optimization" on page 10-2 | "Generate Reusable Code from Library Subsystems Shared Across Models" | "Optimize Generated Code by Passing Reusable Subsystem Outputs as Individual Arguments" | "Code Efficiency"

Remove root level I/O zero initialization

Control generation of initialization code for root-level inports and outports set to zero

Model Configuration Pane: Code Generation / Optimization

Description

The **Remove root level I/O zero initialization** parameter specifies whether to generate initialization code for root-level inports and outports set to zero.

This parameter does not apply to root-level inports and outports that you map to:

- A service interface that uses the during-execution or outside-execution communication method.
- A TLC-based storage class created using the Custom Storage Class Designer.

Settings

on | off

On

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

During startup, standards-compliant C and C++ compilers initialize global data to zero, eliminating the need to include zero initialization code for this data in the generated code. Standards-compliant compilers do not necessarily initialize dynamically allocated data and local variables to zero. Before leaving the **Remove root level I/O zero initialization** parameter selected, confirm that your model meets these conditions:

- If your compiler is not standards-compliant, confirm that it initializes global data to zero.
- If you set the **Code Interface packaging** parameter to **Reusable** function, confirm that data is either statically allocated or that dynamically allocated data is initialized to zero.

Generated code initializes data that you specify in an Initialize Function block, even if you select this parameter.

Off

Generates initialization code for root-level inports and outports.

Generated code does not initialize data whose storage class has imported scope, unless you specify that data in an Initialize Function block. Generated code does not initialize inports that you connect to Initialize Function blocks. Generated code does not initialize data that you map to a service interface that uses the outside-execution or during-execution data communication methods.

If you set the **Code interface packaging** model configuration parameter to **Nonreusable** function, by default, the **Remove root level I/O zero initialization** check box is selected. At the command line, `ZeroExternalMemoryAtStartup` is set to off.

If you set the **Code interface packaging** parameter to **Reusable** function, by default, the **Remove root level I/O zero initialization** check box is cleared. At the command line, `ZeroExternalMemoryAtStartup` is set to on.

If you set the **Code interface packaging** parameter to C++ Class, the **Remove root level I/O zero initialization** check box is selected and cannot be changed. At the command line, ZeroExternalMemoryAtStartup is set to off.

If you set the **Code interface packaging** parameter to Reusable function and select the Use dynamic memory allocation for model initialization parameter, the **Remove root level I/O zero initialization** check box is cleared and ZeroExternalMemoryAtStartup is set to on.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (GUI), off (command line) (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

Programmatic Use

Parameter: ZeroExternalMemoryAtStartup

Type: character vector

Value: 'off' (UI Setting On) | 'on' (UI Setting Off)

Default: 'off'

Version History

Introduced in R2007a

See Also

Topics

“Remove Initialization Code from Root-Level Inports and Outports Set to Zero”

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

“Code Efficiency”

Remove internal data zero initialization

Control generation of Initialization code for internal work structures, such as block states and block outputs, to zero

Model Configuration Pane: Code Generation / Optimization

Description

The **Remove internal data zero initialization** parameter specifies whether to generate initialization code for internal work structures, such as block states and block outputs, to zero.

This parameter does not apply to internal work structures that you map to a TLC-based storage class created using the Custom Storage Class Designer.

Settings

on | off

On

Does not generate code that initializes internal work structures to zero.

During startup, standards-compliant C and C++ compilers initialize global data to zero, eliminating the need to include zero initialization code for this data in the generated code. Standards-compliant compilers do not necessarily initialize dynamically allocated data and local variables to zero. Before leaving the **Remove internal data zero initialization** parameter selected, confirm that your model meets these conditions:

- If your compiler is not standards-compliant, confirm that it initializes global data to zero.
- If you set the **Code interface packaging** to **Reusable** function, confirm that data is either statically allocated or that dynamically allocated data is initialized to zero.

Generated code initializes data that you specify in an Initialize Function block, even if you select this parameter.

Off

Generates code that initializes internal work structures to zero.

Generated code does not initialize data whose storage class has imported scope, unless you specify that data in an Initialize Function block.

If you set the **Code interface packaging** parameter to **Nonreusable** function, by default, the **Remove internal data zero initialization** check box is selected and `ZeroInternalMemoryAtStartup` is set to off.

If you set the **Code interface packaging** parameter to **Reusable** function, by default, the **Remove internal data zero initialization** check box is cleared and `ZeroInternalMemoryAtStartup` is set to on.

If you set the **Code interface packaging** parameter to **C++ Class**, the **Remove internal data zero initialization** check box is selected and cannot be changed. `ZeroInternalMemoryAtStartup` is set to off.

If you set the **Code interface packaging** parameter to **Reusable** function and select the Use dynamic memory allocation for model initialization parameter, the parameter check box is cleared (ZeroInternalMemoryAtStartup is set to on).

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (GUI), off (command line), (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

Programmatic Use

Parameter: ZeroInternalMemoryAtStartup

Type: character vector

Value: 'off' (UI Setting On) | 'on' (UI Setting Off)

Default: 'off'

Version History

Introduced in R2007a

See Also

Topics

"Remove Zero Initialization Code for Internal Data"

"Model Configuration Parameters: Code Generation Optimization" on page 10-2

"Code Efficiency"

Level

Optimization level for generated code

Model Configuration Pane: Code Generation / Optimization

Description

The **Level** parameter specifies the optimization level to apply to the generated code.

Settings

Maximum (default) | Minimum (Debugging) | Balanced with Readability

Minimum (Debugging)

Configure code generation settings for debugging.

Balanced with Readability

Apply code generation optimizations that balance RAM efficiency and execution speed with the readability of the generated code. For example, selecting this value disables optimizations that cross atomic subsystem boundaries.

Maximum

Configure code generation settings based on your code efficiency objectives. Choosing this setting enables the **Priority** parameter. Set the **Priority** parameter to one of these values:

- Balance RAM and speed (default setting)
- Maximum execution speed
- Minimize RAM

Tips

For each **Priority** and **Level** parameter value, there are corresponding values for the parameters in the **Details** section. These are some important differences among these various settings:

- If you set the **Level** parameter to Minimum (debugging), the parameters in the **Details** section are set to off. The code generator does not implement optimizations that remove variables or code making it easier to debug the generated code.
- The parameter settings for Balanced with Readability and Balance RAM and speed are the same except for these three parameters:
 - **Reuse buffers of different sizes and dimensions**
 - **Optimize global data access**
 - **Optimize block operation order in generated code**

The above optimizations can potentially hurt readability because they cross atomic subsystem boundaries and **Optimize block operation order in generated code** might change the block execution order in the generated code so that it is different than in simulation.

- If you have limited RAM, choose the Minimize RAM setting. This setting enables these optimizations that reduce RAM at the expense of a potential slow-down in execution speed:

- **Pack Boolean data into bitfields**
- **Reuse buffers of different sizes and dimensions**
- **Use bitsets for storing state configuration**
- **Use bitsets for storing Boolean data**

This setting also changes the **Optimize block operation order in generated code** from Improved Code Execution Speed to off.

For each **Priority** and **Level** parameter value, this table lists the corresponding values for the parameters in the **Details** section.

Parameters	Settings					Example
Level	Minimum (debugging)	Balanced with readability	Maximum			
Priority	Not Applicable (N/A)	N/A	Balance RAM and speed	Maximize execution speed	Minimize RAM	
Details						
Use memcpy for vector assignment	Off	On	On	On	On	“Use memcpy Function to Optimize Generated Code for Vector Assignment s”
Memcpy threshold (bytes)	Off	64	64	64	64	“Use memcpy Function to Optimize Generated Code for Vector Assignment s”
Enable local block outputs	Off	On	On	On	On	“Enable and Reuse Local Block Outputs in Generated Code”

Parameters	Settings					Example
Reuse local block outputs	Off	On	On	On	On	“Enable and Reuse Local Block Outputs in Generated Code”
Eliminate superfluous local variables (expression folding)	Off	On	On	On	On	Eliminate superfluous local variables (Expression folding)
Reuse global block outputs	Off	On	On	On	On	Reuse global block outputs
Perform in-place updates for Assignment and Bus Assignment blocks	Off	On	on	On	On	Perform in-place updates for Assignment and Bus Assignment blocks
Reuse buffers for Data Store Read and Data Store Write blocks	Off	On	On	On	On	Reuse buffers for Data Store Read and Data Store Write blocks
Simplify array indexing	Off	Off	Off	On	Off	Simplify array indexing
Pack Boolean data into bitfields	Off	Off	Off	Off	On	“Optimize Generated Code by Packing Boolean Data into Bitfields”
Reuse buffers of different sizes and dimensions	Off	Off	On	Off	On	“Reuse Buffers of Different Sizes and Dimensions”

Parameters	Settings					Example
Generate parallel for-loops	Off	Off	Off	On	Off	“Generate Parallel for-Loops Using the Open Multiprocessing (OpenMP) Application Interface”
Automatically schedule for-loops	Off	Off	Off	On	Off	“Automatically Schedule for-Loops for Neighborhood Processing Subsystems”
Reuse buffers for Model blocks	Off	Off	On	On	On	“Reduce Memory Usage for Models Containing Referenced Models”
Unify buffer reuse candidates	Off	Off	Off	Off	Off	“Generate Efficient Code Using Unified Analysis”
Optimize global data access	None	None	Use global to hold temporary results	None	Use global to hold temporary results	“Optimize Global Variable Usage”
Optimize block operation order in generated code	Off	Off	Improved Code Execution Speed	Improved Code Execution Speed	Off	Optimize block operation order in generated code
Use bitsets for storing state configuration	Off	Off	Off	Off	On	Use bitsets for storing state configuration

Parameters	Settings					Example
Use bitsets for storing Boolean data	Off	Off	Off	Off	On	Use bitsets for storing Boolean data

If you plan on upgrading your software, be aware that:

- Setting the **Level** and **Priority** parameters enables the latest optimizations corresponding with the above parameter settings for each subsequent release.
- Selecting the **Specify custom optimizations** parameter enables you to select individual parameters in the **Details** section. When you load a model in a future release, optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to off. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Recommended Settings

Application	Setting
Debugging	Minimum (debugging)
Traceability	Minimum (debugging)
Efficiency	Based on your goals, choose Balanced with Readability or Maximum . If you choose Maximum , set the Priority parameter.
Safety precaution	No impact

Programmatic Use

Parameter: OptimizationLevel

Type character vector

Value: 'level0' (UI Setting Minimum (debugging))| 'level1' (UI Setting Balanced with Readability)| 'level2' (UI Setting Maximum)

Default: 'level2'

Version History

Introduced in R2007b

See Also

Priority | Specify custom optimizations

Topics

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Priority

Control application of optimizations based on priority

Model Configuration Pane: Code Generation / Optimization

Description

The **Priority** parameter optimizes the generated code for increased execution efficiency, decreased RAM consumption, or a balance between execution efficiency and RAM consumption.

Dependencies

This parameter is enabled by setting the **Level** parameter to **Maximum**.

Settings

Balance RAM and speed (default) | Maximize execution speed | Minimize RAM

Balance RAM and speed

Configure code generation settings to balance RAM and execution speed.

Maximize execution speed

Apply code generation settings to maximize execution speed.

Minimize RAM

Configure code generation settings to minimize RAM consumption.

Tips

For each **Priority** and **Level** parameter value, there are corresponding values for the parameters in the **Details** section. These are some important differences among these various settings:

- If you set the **Level** parameter to **Minimum (debugging)**, the parameters in the **Details** section are set to off. The code generator does not implement optimizations that remove variables or code making it easier to debug the generated code.
- The parameter settings for **Balanced with Readability** and **Balance RAM and speed** are the same except for these three parameters:
 - **Reuse buffers of different sizes and dimensions**
 - **Optimize global data access**
 - **Optimize block operation order in generated code**

The above optimizations can potentially hurt readability because they cross atomic subsystem boundaries and **Optimize block operation order in generated code** might change the block execution order in the generated code so that it is different than in simulation.

- If you have limited RAM, choose the **Minimize RAM** setting. This setting enables these optimizations that reduce RAM at the expense of a potential slow-down in execution speed:
 - **Pack Boolean data into bitfields**

- Reuse buffers of different sizes and dimensions
- Use bitsets for storing state configuration
- Use bitsets for storing Boolean data

This setting also changes the **Optimize block operation order in generated code** from Improved Code Execution Speed to off.

For each **Priority** and **Level** parameter value, this table lists the corresponding values for the parameters in the **Details** section.

Parameters	Settings					Example
Level	Minimum (debugging)	Balanced with readability	Maximum			
Priority	Not Applicable (N/A)	N/A	Balance RAM and speed	Maximize execution speed	Minimize RAM	
Details						
Use memcpy for vector assignment	Off	On	On	On	On	“Use memcpy Function to Optimize Generated Code for Vector Assignment s”
Memcpy threshold (bytes)	Off	64	64	64	64	“Use memcpy Function to Optimize Generated Code for Vector Assignment s”
Enable local block outputs	Off	On	On	On	On	“Enable and Reuse Local Block Outputs in Generated Code”
Reuse local block outputs	Off	On	On	On	On	“Enable and Reuse Local Block Outputs in Generated Code”

Parameters	Settings					Example
Eliminate superfluous local variables (expression folding)	Off	On	On	On	On	Eliminate superfluous local variables (Expression folding)
Reuse global block outputs	Off	On	On	On	On	Reuse global block outputs
Perform in-place updates for Assignment and Bus Assignment blocks	Off	On	on	On	On	Perform in-place updates for Assignment and Bus Assignment blocks
Reuse buffers for Data Store Read and Data Store Write blocks	Off	On	On	On	On	Reuse buffers for Data Store Read and Data Store Write blocks
Simplify array indexing	Off	Off	Off	On	Off	Simplify array indexing
Pack Boolean data into bitfields	Off	Off	Off	Off	On	"Optimize Generated Code by Packing Boolean Data into Bitfields"
Reuse buffers of different sizes and dimensions	Off	Off	On	Off	On	"Reuse Buffers of Different Sizes and Dimensions"

Parameters	Settings					Example
Generate parallel for-loops	Off	Off	Off	On	Off	"Generate Parallel for-Loops Using the Open Multiprocessing (OpenMP) Application Interface"
Automatically schedule for-loops	Off	Off	Off	On	Off	"Automatically Schedule for-Loops for Neighborhood Processing Subsystems"
Reuse buffers for Model blocks	Off	Off	On	On	On	"Reduce Memory Usage for Models Containing Referenced Models"
Unify buffer reuse candidates	Off	Off	Off	Off	Off	"Generate Efficient Code Using Unified Analysis"
Optimize global data access	None	None	Use global to hold temporary results	None	Use global to hold temporary results	"Optimize Global Variable Usage"
Optimize block operation order in generated code	Off	Off	Improved Code Execution Speed	Improved Code Execution Speed	Off	Optimize block operation order in generated code
Use bitsets for storing state configuration	Off	Off	Off	Off	On	Use bitsets for storing state configuration

Parameters	Settings					Example
Use bitsets for storing Boolean data	Off	Off	Off	Off	On	Use bitsets for storing Boolean data

If you plan on upgrading your software, be aware that:

- Setting the **Level** and **Priority** parameters enables the latest optimizations corresponding with the above parameter settings for each subsequent release.
- Selecting the **Specify custom optimizations** parameter enables you to select individual parameters in the **Details** section. When you load a model in a future release, optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to off. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Minimize RAM (RAM), Maximum execution speed (Speed), Balance RAM and speed (Balanced)
Safety precaution	No impact

Programmatic Use

Parameter: OptimizationPriority

Type character vector

Value: 'Balanced' | 'Speed' | 'RAM'

Default: 'Balanced'

Version History

Introduced in R2018a

See Also

Level | Specify custom optimizations

Topics

"Model Configuration Parameters: Code Generation Optimization" on page 10-2

Specify custom optimizations

Enable option to choose individual optimization parameters

Model Configuration Pane: Code Generation / Optimization

Description

This **Specify custom optimizations** parameter enables the optimization parameters in the **Details** section and provides you an option to choose individual optimization parameters rather than setting the **Priority** and **Level**.

Dependencies

This parameter enables parameters in the **Details** section.

Settings

off (default) | on

On

Enables you to individually select or clear parameters in the **Details** section.

Off

Disables the parameters in the **Details** section, so that you cannot individually select or clear these parameters.

Clearing this parameter enables the **Level** and **Priority** parameters and resets the parameters in the **Details** section to the current settings for the **Level** and **Priority** parameters.

Tips

If you plan on upgrading your software, be aware that:

- Setting the **Priority** and **Level** parameters enables the latest optimizations corresponding with these settings for each subsequent release.
- Selecting **Specify custom optimizations** means that when you load a model in a future release, optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to off. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Recommended Settings

Application	Setting
Debugging	Off and set Level parameter to Minimize (debugging)
Traceability	Off and set Level parameter to Minimize (debugging)

Application	Setting
Efficiency	Depends on individual parameter settings
Safety precaution	No impact

Programmatic Use

Parameter: OptimizationCustomize

Type character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2018a

See Also

Level | Priority

Topics

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Reuse global block outputs

Optimization to reuse global variables

Model Configuration Pane: Code Generation / Optimization

Description

This **Reuse global block outputs** parameter instructs the code generator to reuse global memory for block outputs whenever possible.

Dependencies

This parameter is enabled by Signal storage reuse.

Settings

on (default) | off

On

The code generator reuses signal memory whenever possible, reducing global variable use.

Off

Signals are stored in unique locations.

Examples

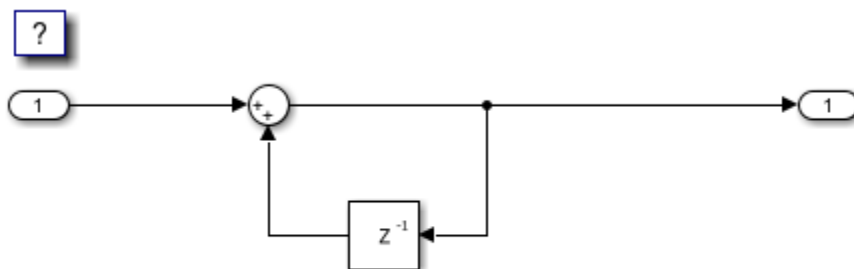
Reuse Global Block Outputs in the Generated Code

Reduce ROM and RAM consumption and data copies and increase execution speed of generated code. Configure the code generator to reuse global variables by selecting the model configuration parameter **Reuse global block outputs**.

Example Model

In the Command Window, open the GlobalReuse model.

```
model='GlobalReuse';  
open_system('GlobalReuse')
```



Copyright 2022 The MathWorks, Inc.

Generate Code Without Optimization

- 1 On the Configuration Parameters dialog box, verify that **Signal storage reuse** is selected.
- 2 On the **Code Generation > Report** pane, verify that **Generate static code metrics** is selected.
- 3 Clear **Reuse global block outputs** and click **Apply**. Alternatively, you can programmatically set this parameter:

```
set_param(model, 'GlobalBufferReuse', 'off');
```

Press **Ctrl+B** to generate code.

```
slbuild(model);

### Searching for referenced models in model 'GlobalReuse'.
### Total of 1 models to build.
### Starting build procedure for: GlobalReuse
### Successful completion of build procedure for: GlobalReuse
```

Build Summary

Top model targets:

Model	Build Reason	Status
GlobalReuse	Information cache folder or artifacts were missing.	Code generated and compiled.

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 17.061s

View the generated code without the optimization. Here is a portion of GlobalReuse.c.

```
cfile = fullfile('GlobalReuse_ert_rtw', 'GlobalReuse.c');
coder.example.extractLines(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void GlobalReuse_step(void)
{
    /* Sum: '<Root>/Sum' incorporates:
     * Delay: '<Root>/Delay'
     * Inport: '<Root>/In1'
     */
    rtDW.Delay_DSTATE += rtU.In1;

    /* Output: '<Root>/Out1' incorporates:
     * Delay: '<Root>/Delay'
     */
    rtY.Out1 = rtDW.Delay_DSTATE;
}
```

The generated code contains a data copy to the global variable `rtDW.Delay_DSTATE`. Open the Static Code Metrics Report. The total number of reads and writes for global variables is 8. The total size is 32 bytes.

Generate Code With Optimization

On the Configuration Parameters dialog box, select **Reuse global block outputs** and click **Apply**. Alternatively, you can programmatically set this parameter:

```
set_param(model, 'GlobalBufferReuse', 'on');

Generate code.

slbuild(model);

### Searching for referenced models in model 'GlobalReuse'.
### Total of 1 models to build.
### Starting build procedure for: GlobalReuse
### Successful completion of build procedure for: GlobalReuse
```

```
Build Summary

Top model targets:

Model          Build Reason          Status          Build Duration
=====
GlobalReuse    Generated code was out of date.  Code generated and compiled.  0h 0m 12.115s

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.233s
```

View the generated code with the optimization. Here is a portion of GlobalReuse.c.

```
cfile = fullfile('GlobalReuse_ert_rtw','GlobalReuse.c');
coder.example.extractLines(cfile,'/* Model step','/* Model initialize',1, 0);

/* Model step function */
void GlobalReuse_step(void)
{
    /* Sum: '<Root>/Sum' incorporates:
     * Delay: '<Root>/Delay'
     * Inport: '<Root>/In1'
     */
    rtY.Out1 += rtU.In1;
}
```

The code generator eliminates a data copy, reduces two statements to one statement and three global variables to two global variables.

Open the Static Code Metrics Report. For global variables, this optimization reduces the total number of reads and writes for global variables from 8 to 5 and the total size from 32 bytes to 24 bytes.

```
bdclose(model)
```

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM, RAM)
Safety precaution	No impact

Programmatic Use

Parameter: GlobalBufferReuse

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2014a

See Also

Topics

“Code Efficiency”

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Perform in-place updates for Assignment and Bus Assignment blocks

Reuse the input and output variables of Bus Assignment and Assignment blocks

Model Configuration Pane: Code Generation / Optimization

Description

The **Perform in-place updates for Assignment and Bus Assignment blocks** parameter instructs the code generator to reuse the input and output variables of Bus Assignment and Assignment blocks if possible.

Dependencies

The parameter **Signal Storage Reuse** enables this parameter.

Settings

on (default) | off

On

Embedded Coder reuses the input and output variables of Bus Assignment and Assignment blocks if possible. Reusing these variables reduces data copies, conserves RAM consumption and increases code execution speed.

Off

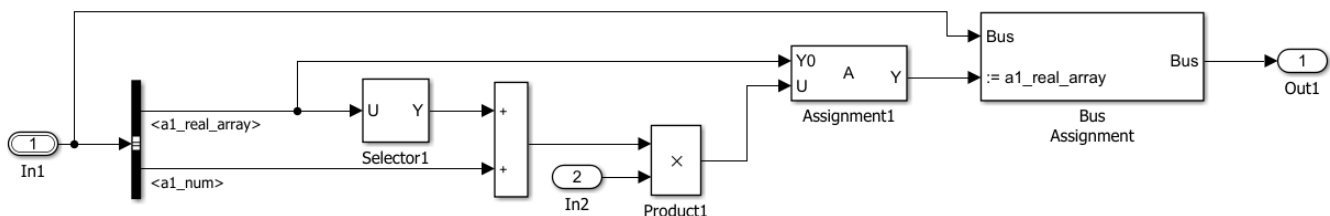
Embedded Coder does not reuse the input and output variables of Bus Assignment and Assignment blocks.

Examples

Reduce Data Copies for Bus Assignment Blocks in Generated Code

For models containing a Bus Assignment block, if possible, the code generator uses the same variable for the block input and output. Reusing these variables reduces data copies, conserves RAM consumption and increases code execution speed.

For example, in `bus_assignoptim`, a bus signal containing six elements feeds into a Bus Assignment block and a Bus Selector block. The Bus Assignment block assigns new values to the bus element `a1_real_array`. This bus signal feeds into Out1.



Generate code without the optimization by setting the **Perform in-place updates for Assignment and Bus Assignment blocks** parameter to off. Without the optimization the `bus_assignoptim_step` function contains this code:

```
void bus_assignoptim_step(void)
{
    real_T rtb_Assignment[36];
    int32_T i;

    /* Assignment: '<Root>/Assignment' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Product: '<Root>/Product'
     * Selector: '<Root>/Selector'
     * Sum: '<Root>/Sum1'
     */
    for (i = 0; i < 36; i++) {
        rtb_Assignment[i] = bus_assignoptim_U.In1.al_real_array[i];
    }

    for (i = 0; i < 2; i++) {
        rtb_Assignment[(int32_T)(i + 22)] = (bus_assignoptim_U.In1.al_real_array
            [(int32_T)(i + 22)] + bus_assignoptim_U.In1.al_num) *
            bus_assignoptim_U.In2;
    }

    /* End of Assignment: '<Root>/Assignment' */

    /* Output: '<Root>/Out' incorporates:
     * BusAssignment: '<Root>/Bus Assignment'
     * Inport: '<Root>/In1'
     */
    bus_assignoptim_Y.Out = bus_assignoptim_U.In1;

    /* BusAssignment: '<Root>/Bus Assignment' incorporates:
     * Output: '<Root>/Out'
     */
    for (i = 0; i < 36; i++) {
        bus_assignoptim_Y.Out.al_real_array[i] = rtb_Assignment[i];
    }
}
```

The generated code contains the temporary array `rtb_Assignment` for holding data before this data is assigned to `bus_assignoptim_Y.Out.al_real_array`.

Generate code with the optimization by setting the **Perform in-place updates for Assignment and Bus Assignment blocks** parameter to on. With the optimization, the `bus_assignoptim_step` function contains this code:

```
/* Model step function */
void bus_assignoptim_step(void)
{
    int32_T i;

    /* Output: '<Root>/Out' incorporates:
     * Inport: '<Root>/In1'
     * SignalConversion: '<Root>/TmpBusAssignmentBufferAtBus AssignmentInport1'
     */
}
```

```
bus_assignoptim_Y.Out = bus_assignoptim_U.In1;

/* Assignment: '<Root>/Assignment' incorporates:
 * Inport: '<Root>/In1'
 * Inport: '<Root>/In2'
 * Outport: '<Root>/Out'
 * Product: '<Root>/Product'
 * Selector: '<Root>/Selector'
 * Sum: '<Root>/Sum1'
 */
for (i = 0; i < 36; i++) {
    bus_assignoptim_Y.Out.al_real_array[i] =
        bus_assignoptim_U.In1.al_real_array[i];
}

for (i = 0; i < 2; i++) {
    bus_assignoptim_Y.Out.al_real_array[(int32_T)(i + 22)] =
        (bus_assignoptim_U.In1.al_real_array[(int32_T)(i + 22)] +
         bus_assignoptim_U.In1.al_num) * bus_assignoptim_U.In2;
}

/* End of Assignment: '<Root>/Assignment' */
}
```

The generated code does not contain the temporary array `rtb_Assignment1` for holding data. The generated code directly assigns this data to `bus_assignoptim_Y.Out.al_real_array`.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

Programmatic Use

Parameter: BusAssignmentInplaceUpdate
Type: character vector
Value: 'on' | 'off'
Default: 'on'

Version History

Introduced in R2017a

See Also

Pass reusable subsystem outputs as

Topics

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Reuse buffers for Data Store Read and Data Store Write blocks

Eliminate temporary buffers for Data Store Read and Data Store Write blocks

Model Configuration Pane: Code Generation / Optimization

Description

The **Reuse buffers for Data Store Read and Data Store Write blocks** parameter instructs the code generator to remove temporary buffers for Data Store Read and Data Store Write blocks. Use the Data Store Memory block directly if possible.

Dependencies

The parameter **Signal Storage Reuse** enables this parameter.

Settings

on (default) | off

On

Embedded Coder reads directly from the Data Store Memory block and writes directly to the Data Store Memory block, if possible. Using the Data Store Memory block directly eliminates data copies in the generated code, conserving RAM consumption and increasing code execution speed.

Off

Embedded Coder inserts buffers in the generated code for Data Store Read and Data Store Write blocks.

Examples

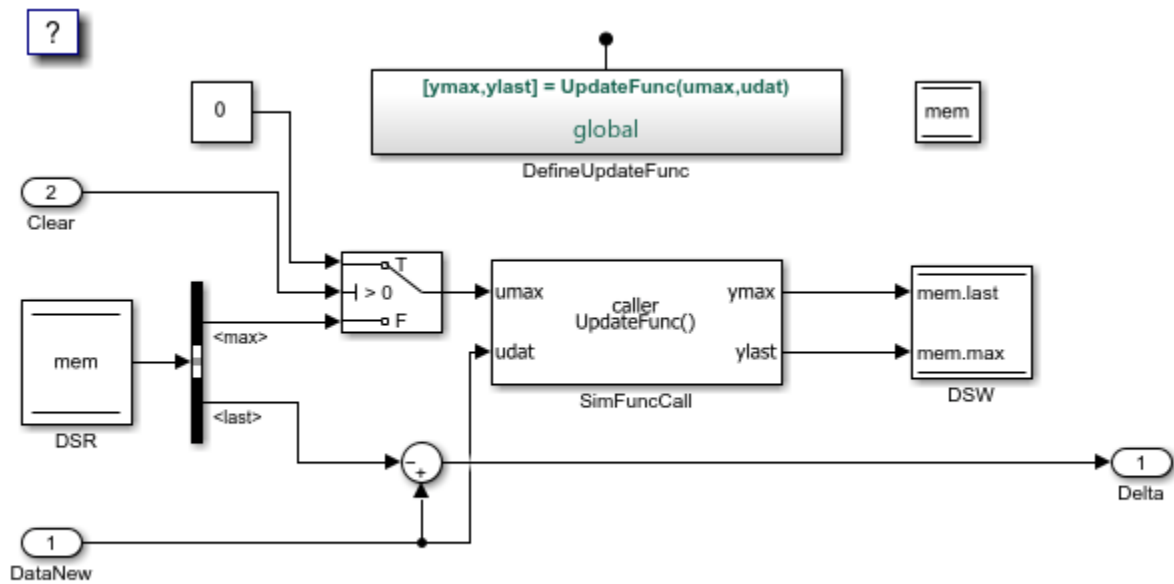
Data Copy Reduction for Data Store Read and Data Store Write Blocks

This example shows how the code generator removes temporary buffers for Data Store Read and Data Store Write blocks. This optimization improves execution speed and reduces RAM consumption.

Example Model

The model `rtwdemo_optimizedatastorebuffers` contains the Function caller `UpdateFunc`, which calls the Simulink Function `DefineUpdateFunc`. The Data Store Read block `DSR` reads from `mem`. The Data Store Write block `DSW` writes to `mem`.

```
model='rtwdemo_optimizedatastorebuffers';  
open_system(model);
```



Copyright 2016-2023 The MathWorks, Inc.

Generate Code Without Optimization

In the Configuration Parameters dialog box, deselect the **Reuse buffers for Data Store Read and Data Store Write blocks parameter** or at the MATLAB® command prompt, enter this command:

```
set_param(model, 'OptimizeDataStoreBuffers', 'off');
```

Build the model.

```
slbuild(model)
```

```
### Searching for referenced models in model 'rtwdemo_optimizeddatastorebuffers'.
### Total of 1 models to build.
### Starting build procedure for: rtwdemo_optimizeddatastorebuffers
### Successful completion of build procedure for: rtwdemo_optimizeddatastorebuffers
```

Build Summary

Top model targets:

Model	Build Reason	Status
rtwdemo optimizeddatastorebuffers	Information cache folder or artifacts were missing.	Code generated

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 58.428s
```

View the generated code without the optimization. This code is in `rtwdemo_optimizedatastorebuffers.c`.

```
cfile = fullfile('rtwdemo_optimizeddatastorebuffers_ert_rtw',...
    'rtwdemo_optimizeddatastorebuffers.c');
coder.example.extractLines(cfile, '/* Model step', '/* Model initialize', 1, 0);
```

```

/* Model step function */
void step(void)
{
    real_T rtb_DSR_last;
    real_T rtb_SimFuncCall_o1;
    real_T rtb_Sum_p;

    /* DataStoreRead: '<Root>/DSR' */
    rtb_DSR_last = mem.last;

    /* Switch: '<Root>/Switch' incorporates:
     * Constant: '<Root>/Constant'
     * DataStoreRead: '<Root>/DSR'
     * Inport: '<Root>/Clear'
     */
    if (rtU.Clear) {
        rtb_SimFuncCall_o1 = 0.0;
    } else {
        rtb_SimFuncCall_o1 = mem.max;
    }

    /* End of Switch: '<Root>/Switch' */

    /* FunctionCaller: '<Root>/SimFuncCall' incorporates:
     * Inport: '<Root>/DataNew'
     */
    UpdateFunc(rtb_SimFuncCall_o1, rtU.DataNew, &rtb_SimFuncCall_o1, &rtb_Sum_p);

    /* DataStoreWrite: '<Root>/DSW' */
    mem.last = rtb_SimFuncCall_o1;
    mem.max = rtb_Sum_p;

    /* Outport: '<Root>/Delta' incorporates:
     * DataStoreRead: '<Root>/DSR'
     * Inport: '<Root>/DataNew'
     * Sum: '<Root>/Sum'
     */
    rtY.Delta = rtU.DataNew - rtb_DSR_last;
}

```

The generated code contains data copies for the Data Store Read and Data Store Write blocks, respectively.

Generate Code with Optimization

In the Configuration Parameters dialog box, clear the **Reuse buffers for Data Store Read and Data Store Write blocks** parameter or at the MATLAB command prompt, enter this command:

```
set_param(model, 'OptimizeDataStoreBuffers', 'on');
```

Build the model.

```
slbuild(model)
```

```

### Searching for referenced models in model 'rtwdemo_optimizedatastorebuffers'.
### Total of 1 models to build.
### Starting build procedure for: rtwdemo_optimizedatastorebuffers

```

```
### Successful completion of build procedure for: rtwdemo_optimizedatastorebuffers
```

```
Build Summary
```

```
Top model targets:
```

Model	Build Reason	Status
=====		
rtwdemo_optimizedatastorebuffers	Generated code was out of date.	Code generated and compiled.

```
1 of 1 models built (0 models already up to date)
```

```
Build duration: 0h 0m 17.385s
```

View the generated code with the optimization. This code is in
`rtwdemo_optimizedatastorebuffers.c`.

```
cfile = fullfile('rtwdemo_optimizedatastorebuffers_ert_rtw',...  
    'rtwdemo_optimizedatastorebuffers.c');  
coder.example.extractLines(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */  
void step(void)  
{  
    real_T rtb_DSR_last;  
    real_T tmp;  
  
    /* DataStoreRead: '<Root>/DSR' */  
    rtb_DSR_last = mem.last;  
  
    /* Switch: '<Root>/Switch' incorporates:  
     * Constant: '<Root>/Constant'  
     * DataStoreRead: '<Root>/DSR'  
     * Inport: '<Root>/Clear'  
     */  
    if (rtU.Clear) {  
        tmp = 0.0;  
    } else {  
        tmp = mem.max;  
    }  
  
    /* FunctionCall: '<Root>/SimFuncCall' incorporates:  
     * DataStoreWrite: '<Root>/DSW'  
     * Inport: '<Root>/DataNew'  
     * Switch: '<Root>/Switch'  
     */  
    UpdateFunc(tmp, rtU.DataNew, &mem.last, &mem.max);  
  
    /* Outport: '<Root>/Delta' incorporates:  
     * DataStoreRead: '<Root>/DSR'  
     * Inport: '<Root>/DataNew'  
     * Sum: '<Root>/Sum'  
     */  
    rtY.Delta = rtU.DataNew - rtb_DSR_last;  
}
```

The data copy for the Data Store Write block is not in the generated code. The code contains the data copy for the Data Store Read block because the Sum block executes after the Data Store Write block.

The generated code contains the variable `rtb_DSR_last` to hold the output of the Sum block. Therefore, the Sum block gets the values that `SimFuncCall` calculates at the start of the time step rather than those values at the next time step. If the priority of the Sum block is lower than `SimFuncCall`, the code generator can remove the data copy for the Data Store Read block. Some other cases in which the code generator might not eliminate data copies are:

- A Simulink Function internally writes to the Data Store Memory block.
- The Data Store Read or Data Store Write blocks select elements of an array from the Data Store Memory block.
- The Data Store Memory block has a custom storage class.
- The Data Store Read and Data Store Write blocks occur on the same block unless that block is a Bus Assignment block or an Assignment block.

Close the model and clean up.

```
bdclose(model)
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On
Safety precaution	No recommendation

Programmatic Use

Parameter: `OptimizeDataStoreBuffers`

Type: character vector

Value: `'on'` | `'off'`

Default: `'on'`

Version History

Introduced in R2017a

See Also

Topics

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Simplify array indexing

Replace multiply operations in array indices when accessing arrays in a loop

Model Configuration Pane: Code Generation / Optimization

Description

The **Simplify array indexing** parameter instructs the code generator whether to replace multiply operations in array indices when accessing arrays in a loop.

Settings

off (default) | on

Default: Off

On

In array indices, replace multiply operations with add operations when accessing arrays in a loop in the generated code. When the original signal is multidimensional, the Embedded Coder generates one-dimensional arrays, resulting in multiply operations in the array indices. Using this setting eliminates costly multiply operations when accessing arrays in a loop in the C/C++ program. This optimization (commonly referred to as strength reduction) is particularly useful if the C/C++ compiler on the target platform does not have similar functionality. The absence of multiply operations in the C/C++ program does not imply that the C/C++ compiler does not generate multiply instructions.

Off

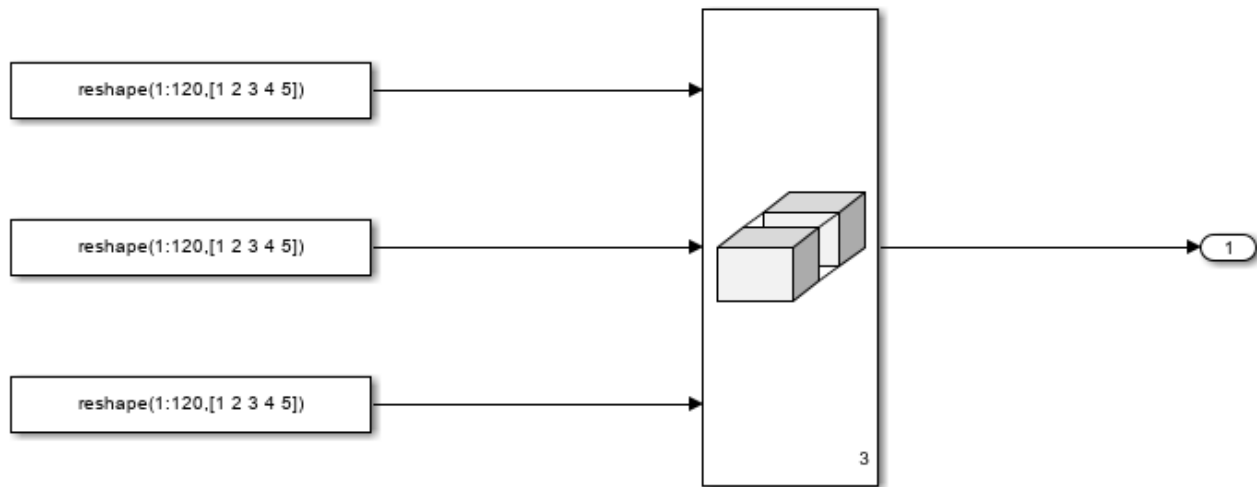
Leave multiply operations in array indices when accessing arrays in a loop.

Examples

Simplify Multiply Operations in Array Indexing

The generated code might have multiply operations when indexing an element of an array. You can select the optimization parameter Simplify array indexing to replace multiply operations in the array index with a temporary variable. This optimization can improve execution speed by reducing the number of times the multiply operation executes.

If you have the following model:



The Constant blocks have the following **Constant value**:

- Const1: reshape(1:120,[1 2 3 4 5])
- Const2: reshape(1:120,[1 2 3 4 5])
- Const3: reshape(1:120,[1 2 3 4 5])

The Concatenate block parameter **Mode** is set to Multidimensional array. The Constant blocks **Sample time** parameter is set to -1.

Building the model with the **Simplify array indexing** parameter turned off generates the following code:

```
real_T pooled1;
int32_T Out1_tmp;
int32_T Out1_tmp_0;
int32_T i;
int32_T i_0;
int32_T i_1;
int32_T pooled1_tmp;
int32_T pooled1_tmp_0;

for (i_1 = 0; i_1 < 5; i_1++) {
    for (i_0 = 0; i_0 < 4; i_0++) {
        for (i = 0; i < 3; i++) {
            pooled1_tmp = i << 1;
            pooled1_tmp_0 = (6 * i_0 + pooled1_tmp) + 24 * i_1;
            pooled1 = ex_arrayindex_ConstP.pooled1[pooled1_tmp_0];
            pooled1_tmp = (18 * i_0 + pooled1_tmp) + 72 * i_1;
            ex_arrayindex_Y.Out1[pooled1_tmp] = pooled1;

            Out1_tmp = (((i + 3) << 1) + 18 * i_0) + 72 * i_1;
            ex_arrayindex_Y.Out1[Out1_tmp] = pooled1;
        }
    }
}
```

```

    Out1_tmp_0 = (((i + 6) << 1) + 18 * i_0) + 72 * i_1;
    ex_arrayindex_Y.Out1[Out1_tmp_0] = pooled1;

    pooled1 = ex_arrayindex_ConstP.pooled1[pooled1_tmp_0 + 1];
    ex_arrayindex_Y.Out1[pooled1_tmp + 1] = pooled1;
    ex_arrayindex_Y.Out1[Out1_tmp + 1] = pooled1;
    ex_arrayindex_Y.Out1[Out1_tmp_0 + 1] = pooled1;
  }
}
}

```

Open the Configuration Parameters dialog box and select the **Simplify array indexing** parameter. Build the model again. In the generated code, $(tmp\_3 + tmp\_2) + tmp\_0$ replaces a multiply operation in the array index. The generated code is now:

```

real_T pooled1;
int32_T Out1_tmp;
int32_T i;
int32_T i_0;
int32_T i_1;
int32_T pooled1_tmp;
int32_T tmp;
int32_T tmp_0;
int32_T tmp_1;
int32_T tmp_2;
int32_T tmp_3;

tmp = 0;
tmp_0 = 0;
for (i_1 = 0; i_1 < 5; i_1++) {
    tmp_1 = 0;
    tmp_2 = 0;
    for (i_0 = 0; i_0 < 4; i_0++) {
        tmp_3 = 0;
        for (i = 0; i < 3; i++) {
            pooled1_tmp = (tmp_3 + tmp_1) + tmp;
            pooled1 = ex_arrayindex_ConstP.pooled1[pooled1_tmp];
            Out1_tmp = (tmp_3 + tmp_2) + tmp_0;
            ex_arrayindex_Y.Out1[Out1_tmp] = pooled1;
            ex_arrayindex_Y.Out1[Out1_tmp + 6] = pooled1;
            ex_arrayindex_Y.Out1[Out1_tmp + 12] = pooled1;
            pooled1 = ex_arrayindex_ConstP.pooled1[pooled1_tmp + 1];
            ex_arrayindex_Y.Out1[Out1_tmp + 1] = pooled1;
            ex_arrayindex_Y.Out1[Out1_tmp + 7] = pooled1;
            ex_arrayindex_Y.Out1[Out1_tmp + 13] = pooled1;
            tmp_3 += 2;
        }

        tmp_1 += 6;
        tmp_2 += 18;
    }

    tmp += 24;
}

```

```
    tmp_0 += 72;  
}
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution speed)
Safety precaution	No impact

Programmatic Use

Parameter: StrengthReduction

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2009a

See Also

Topics

“Code Efficiency”

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Pack Boolean data into bitfields

Control storing Boolean signals

Model Configuration Pane: Code Generation / Optimization

Description

The **Pack Boolean data into bitfields** parameter specifies whether Boolean signals are stored as one-bit bitfields or as a Boolean data type.

Note You cannot use this optimization when you generate code for a target that specifies an explicit structure alignment.

Note This parameter does not apply to Boolean signals assigned to a new storage class in the Embedded Coder Dictionary with the **Use bit fields** property enabled.

Settings

off (default) | on

On

Stores Boolean signals into one-bit bitfields in global block I/O structures or DWork vectors. This will reduce RAM, but might cause more executable code.

Off

Stores Boolean signals as a Boolean data type in global block I/O structures or DWork vectors.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), On (RAM)
Safety precaution	No impact

Programmatic Use

Parameter: BooleansAsBitfields

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2009b

See Also

Topics

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

“Optimize Generated Code by Packing Boolean Data into Bitfields”

Bitfield declarator type specifier

“Code Efficiency”

Bitfield declarator type specifier

Specify bitfield type

Model Configuration Pane: Code Generation / Optimization

Description

The **Bitfield declarator type specifier** parameter specifies the bitfield type when selecting configuration parameter Pack Boolean data into bitfields. The optimization benefit is dependent upon your choice of target.

Dependencies

The **Pack Boolean data into bitfields** parameter enables this parameter.

Settings

`uint_T` (default) | `uchar_T`

`uint_T`

The type specified for a bitfield declaration is an unsigned `int`.

`uchar_T`

The type specified for a bitfield declaration is an unsigned `char`.

Tip

The Pack Boolean data into bitfields configuration parameter default setting uses unsigned integers. This might cause an increase in RAM if the bitfields are small and distributed. In this case, `uchar_T` might use less RAM depending on your target.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Target dependent
Safety precaution	No impact

Programmatic Use

Parameter: BitfieldContainerType

Type: character vector

Value: `uint_T` | `uchar_T`

Default: `uint_T`

Version History

Introduced in R2010a

See Also

Topics

"Model Configuration Parameters: Code Generation Optimization" on page 10-2

"Code Efficiency"

Reuse buffers of different sizes and dimensions

Reuse buffers to store data of different sizes and dimensions

Model Configuration Pane: Code Generation / Optimization

Description

The **Reuse buffers of different sizes and dimensions** parameter specifies to reduce memory consumption by reusing buffers to store data of different sizes and dimensions.

Dependencies

This parameter is enabled by Signal storage reuse.

Settings

on (default) | off

On

The code generator tries to reuse the same buffers to store data of different sizes and dimensions. This optimization conserves RAM and ROM consumption.

Off

The code generator reuses buffers only if they have the same size and shape as the data.

Tips

- If your model contains a reusable custom storage class to specify reuse on signals that have different sizes and shapes, you must select the **Reuse buffers of different sizes and dimensions** parameter or remove the specification. Otherwise, during simulation, the model produces an error.
- The code generator does not replace a buffer with a lower priority buffer that has a smaller size.
- The code generator does not reuse buffers that have different sizes and symbolic dimensions.

Recommended Settings

Application	Setting
Debugging	off
Traceability	off
Efficiency	on
Safety precaution	No impact

Programmatic Use

Parameter: DifferentSizesBufferReuse

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2017b

See Also

Topics

"Model Configuration Parameters: Code Generation Optimization" on page 10-2

"Reuse Buffers of Different Sizes and Dimensions"

Generate parallel for-loops

Optimization for parallel for-loops

Model Configuration Pane: Code Generation / Optimization

Description

Specify, whether for-loops in the generated code are implemented in parallel for the MATLAB Function block, the MATLAB System block, and the For Each Subsystem block with large input data sizes. The MATLAB Function block and MATLAB System block are executed in parallel when the MATLAB code contains `parfor`-loops instead of traditional `for`-loops. This parameter also enables **Automatically schedule for-loops**.

Category: Optimization

Dependencies

This parameter requires that Parallel Computing Toolbox be installed.

Settings

off (default) | on

On

Executes OpenMP parallel for-loops by using multiple threads.

Off

Executes for-loops using a single thread.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution speed)
Safety precaution	No impact

Programmatic Use

Parameter: MultiThreadedLoops

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2019a

See Also

Topics

“Algorithm Acceleration Using Parallel for-Loops (parfor)”

“Generate Parallel for-Loops Using the Open Multiprocessing (OpenMP) Application Interface”

Automatically schedule for-loops

Automatically schedule generated loop nest code

Model Configuration Pane: Code Generation / Optimization

Description

The **Automatically schedule for-loops** parameter specifies whether for-loop nests in the generated code for Neighborhood Processing Subsystem, Pixel Processing Subsystem, and Array Processing Subsystem blocks are automatically scheduled. Automatic scheduling can increase execution speed.

Category: Optimization

Dependencies

To enable this parameter:

- Select the **Specify custom optimizations** model configuration parameter.
- Select the **Generate parallel for-loops** model configuration parameter.
- Clear the **Generate Halide code** model configuration parameter.

Settings

off (default) | on

On

Automatically schedules generated loop nest code.

Off

Does not automatically schedule generated loop nest code.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution speed)
Safety precaution	No impact

Programmatic Use

Parameter: AutoScheduleForLoops

Type: character vector

Values: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2024a

See Also

Generate parallel for-loops

Topics

"Automatically Schedule for-Loops for Neighborhood Processing Subsystems"

Optimize global data access

Optimize global variable usage

Model Configuration Pane: Code Generation / Optimization

Description

The **Optimize global data access** parameter specifies global variable optimization.

Dependencies

This parameter is enabled by **Signal storage reuse**.

Settings

Use global to hold temporary results (default) | None | Minimize global data access

None

Use default optimizations.

Use global to hold temporary results

Maximize use of global variables.

Minimize global data access

Minimize use of global variables by using local variables to hold intermediate values.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	'Use global to hold temporary results' (RAM), 'Minimize global data access' (ROM)
Safety precaution	No impact

Programmatic Use

Parameter: GlobalVariableUsage

Type: character vector

Value: 'None' | 'Use global to hold temporary results' | 'Minimize global data access'

Default: 'Use global to hold temporary results'

Version History

Introduced in R2014a

See Also

Topics

“Optimize Global Variable Usage”

“Code Efficiency”

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Optimize block operation order in generated code

Specify to reorder block operations for improved code execution speed

Model Configuration Pane: Code Generation / Optimization

Description

The **Optimize block operation order in generated code** parameter specifies to reorder block operations in the generated code for improved code execution speed.

Settings

Improved Code Execution Speed (default) | off

off

Embedded Coder does not reorder block operation order in the generated code to create additional instances of buffer reuse.

Improved Code Execution Speed

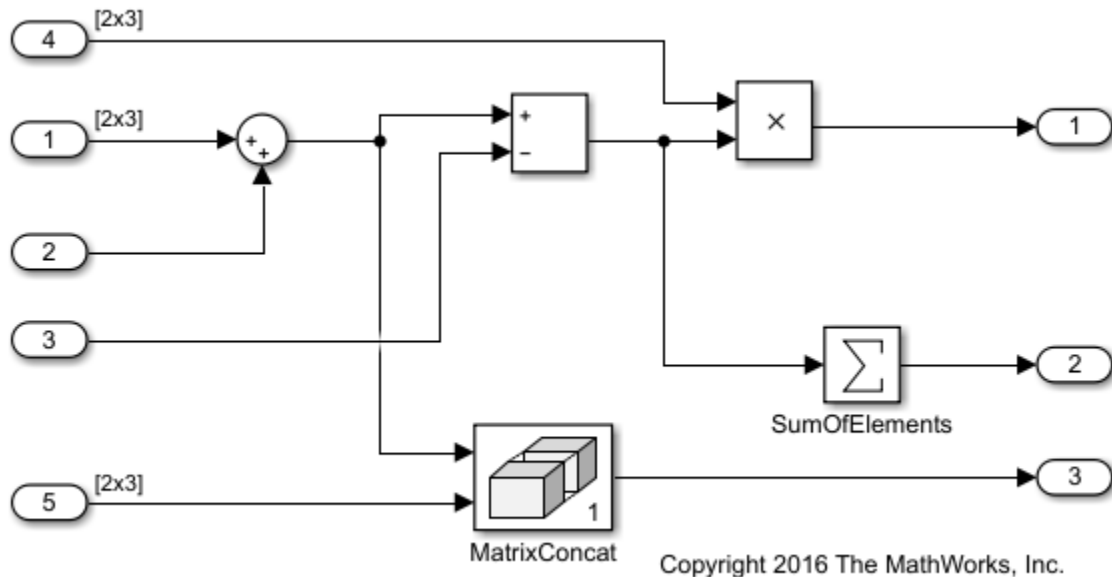
Embedded Coder changes the block operation order in the generated code so that more instances of buffer reuse can occur. Reusing buffers conserves RAM and ROM consumption and improves code execution speed.

Examples

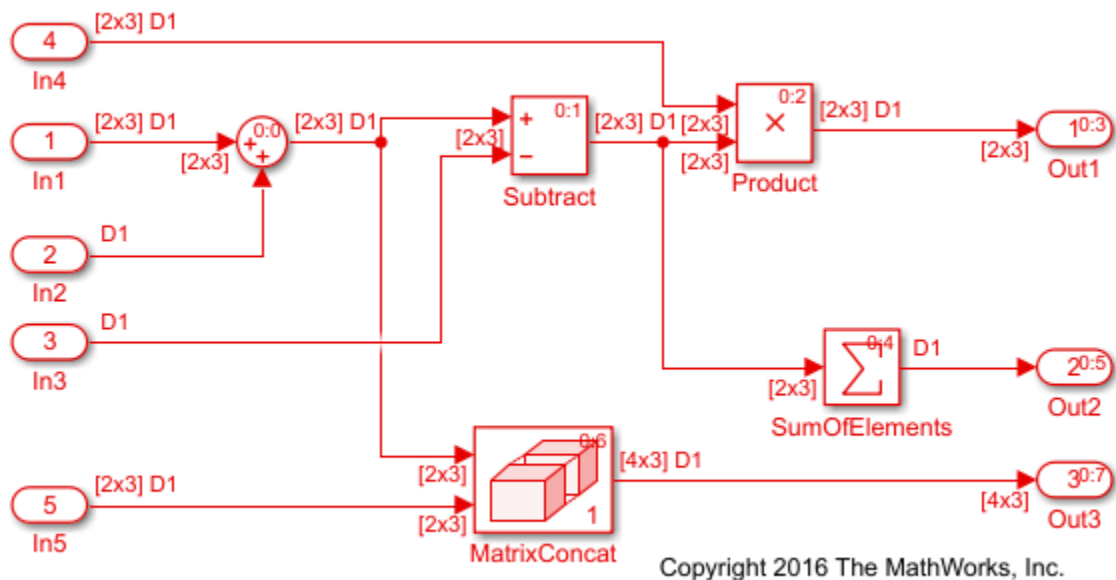
Remove Data Copies by Reordering Block Operations in the Generated Code

This example show how to remove data copies by changing the **Optimize block order in the generated code** parameter from off to Improved Execution Speed. Changing this setting indicates to the code generator to reorder block operations where possible to remove data copies. This optimization conserves RAM and ROM consumption.

In the model `ex_optimizeblockorder`, the signal that leaves the Sum block enters a Subtract block and a Concatenate block. The signal that leaves the Subtract block enters a Product block and a Sum of Elements block.



The image shows the model `ex_optimizeblockorder` after a model build. The red numbers indicate the default block order in the generated code. The Subtract block executes before the Concatenate block. The Product block executes before the Sum of Elements block.



View the generated code without the optimization. Here is the `ex_optimizeblockorder_step` function.

```
/* Model step function */
void ex_optimizeblockorder_step(void)
{
```

```
real_T rtb_Sum2x3[6];
int32_T i;
real_T rtb_Sum2x3_d;
real_T rtb_Subtract;

/* Sum: '<Root>/SumOfElements' */
rtY.Out2 = -0.0;
for (i = 0; i < 6; i++) {
    /* Sum: '<Root>/Sum2x3' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     */
    rtb_Sum2x3_d = rtU.In1[i] + rtU.In2;

    /* Sum: '<Root>/Subtract' incorporates:
     * Inport: '<Root>/In3'
     */
    rtb_Subtract = rtb_Sum2x3_d - rtU.In3;

    /* Output: '<Root>/Out1' incorporates:
     * Inport: '<Root>/In4'
     * Product: '<Root>/Product'
     */
    rtY.Out1[i] = rtU.In4[i] * rtb_Subtract;

    /* Sum: '<Root>/Sum2x3' */
    rtb_Sum2x3[i] = rtb_Sum2x3_d;

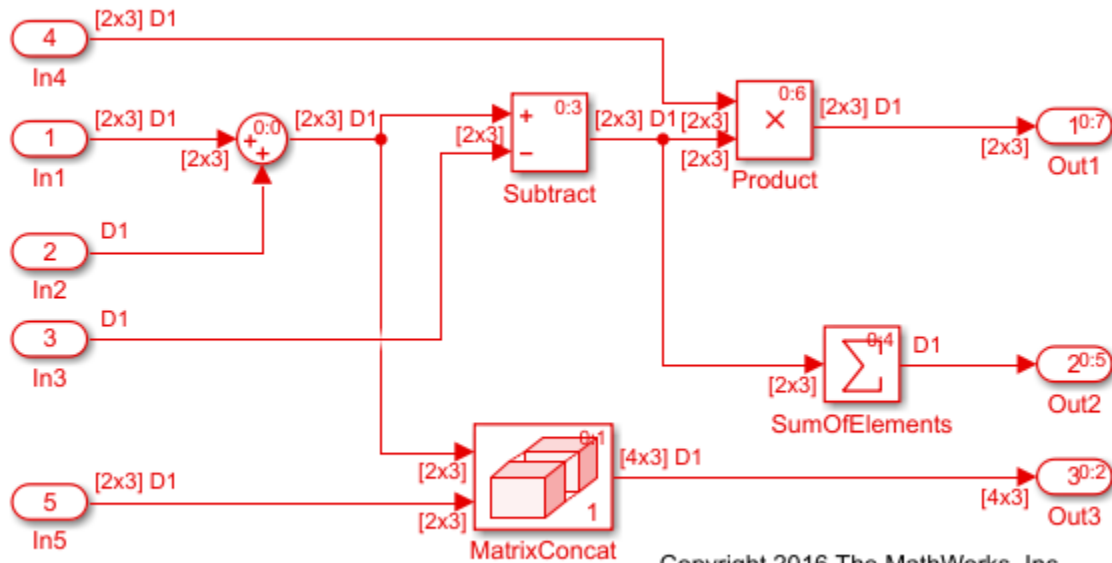
    /* Sum: '<Root>/SumOfElements' */
    rtY.Out2 += rtb_Subtract;
}

/* Concatenate: '<Root>/MatrixConcat ' */
for (i = 0; i < 3; i++) {
    /* Output: '<Root>/Out3' incorporates:
     * Inport: '<Root>/In5'
     */
    rtY.Out3[i << 2] = rtb_Sum2x3[i << 1];
    rtY.Out3[2 + (i << 2)] = rtU.In5[i << 1];
    rtY.Out3[1 + (i << 2)] = rtb_Sum2x3[(i << 1) + 1];
    rtY.Out3[3 + (i << 2)] = rtU.In5[(i << 1) + 1];
}

/* End of Concatenate: '<Root>/MatrixConcat ' */
}
```

With the default order, the generated code contains three buffers, `rtb_Sum2x3[6]`, `rtb_Sum2x3_d`, and `rtb_Subtract`. The generated code contains these temporary variables and associated data copies because the Matrix Concatenate block must use the output from the Sum block and the Sum of Elements block must use the output from the Subtract block.

The image shows the `ex_optimizeblockorder` model after setting the **Optimize block operation order in generated code** parameter to Improved Execution Speed and building the model. The Subtract block executes after the Concatenate block. The Product block executes after the Sum of Elements block.



In the optimized code, the three buffers `rtb_Sum2x3[6]`, `rtb_Sum2x3_d`, and `rtb_Subtract` and their associated data copies are gone. The generated code does not require these temporary variables to hold the outputs of the Sum and Subtract blocks because the Subtract block executes after the Concatenate block and the Product block executes after the Sum of Elements block.

```
/* Model step function */
void ex_optimizeblockorder_step(void)
{
    int32_T i;

    /* Sum: '<Root>/Sum2x3' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     */
    for (i = 0; i < 6; i++) {
        rtY.Out1[i] = rtU.In1[i] + rtU.In2;
    }

    /* End of Sum: '<Root>/Sum2x3' */

    /* Concatenate: '<Root>/MatrixConcat ' */
    for (i = 0; i < 3; i++) {
        /* Outport: '<Root>/Out3' incorporates:
         * Inport: '<Root>/In5'
         */
        rtY.Out3[i << 2] = rtY.Out1[i << 1];
        rtY.Out3[2 + (i << 2)] = rtU.In5[i << 1];
        rtY.Out3[1 + (i << 2)] = rtY.Out1[(i << 1) + 1];
        rtY.Out3[3 + (i << 2)] = rtU.In5[(i << 1) + 1];
    }

    /* End of Concatenate: '<Root>/MatrixConcat ' */
}
```

```
/* Sum: '<Root>/SumOfElements' */
rtY.Out2 = -0.0;
for (i = 0; i < 6; i++) {
  /* Sum: '<Root>/Subtract' incorporates:
   * Inport: '<Root>/In3'
   */
  rtY.Out1[i] -= rtU.In3;

  /* Sum: '<Root>/SumOfElements' */
  rtY.Out2 += rtY.Out1[i];

  /* Outport: '<Root>/Out1' incorporates:
   * Inport: '<Root>/In4'
   * Product: '<Root>/Product'
   */
  rtY.Out1[i] *= rtU.In4[i];
}
}
```

To implement buffer reuse, the code generator does not violate user-specified block priorities.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Improved Code Execution Speed
Safety precaution	No recommendation

Programmatic Use

Parameter: OptimizeBlockOrder
Type: character vector
Value: 'speed' | 'off'
Default: 'speed'

Version History

Introduced in R2017a

See Also

Topics

“Improve Execution Efficiency by Reordering Block Operations in the Generated Code”
“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Optimize using the specified minimum and maximum values

Optimize using minimum and maximum values for signals and parameters

Model Configuration Pane: Code Generation / Optimization

Description

The **Optimize using the specified minimum and maximum values** parameter instructs the code generator whether to optimize generated code using the specified minimum and maximum values for signals and parameters in the model.

The specified minimum and maximum values usually represent environmental limits, such as temperature, or mechanical and electrical limits, such as output ranges of sensors. This optimization uses these values to streamline the generated code by:

- Reducing expressions to constants
- Removing dead branches of conditional statements
- Eliminating unnecessary mathematical operations

Note Make sure the minimum and maximum values that you specify are valid limits. Otherwise, this optimization might result in numerical mismatch with simulation.

Settings

off (default) | on

On

Optimizes the generated code using range information derived from the minimum and maximum specified values for signals and parameters in the model.

Off

Ignores specified minimum and maximum values when generating code.

Examples

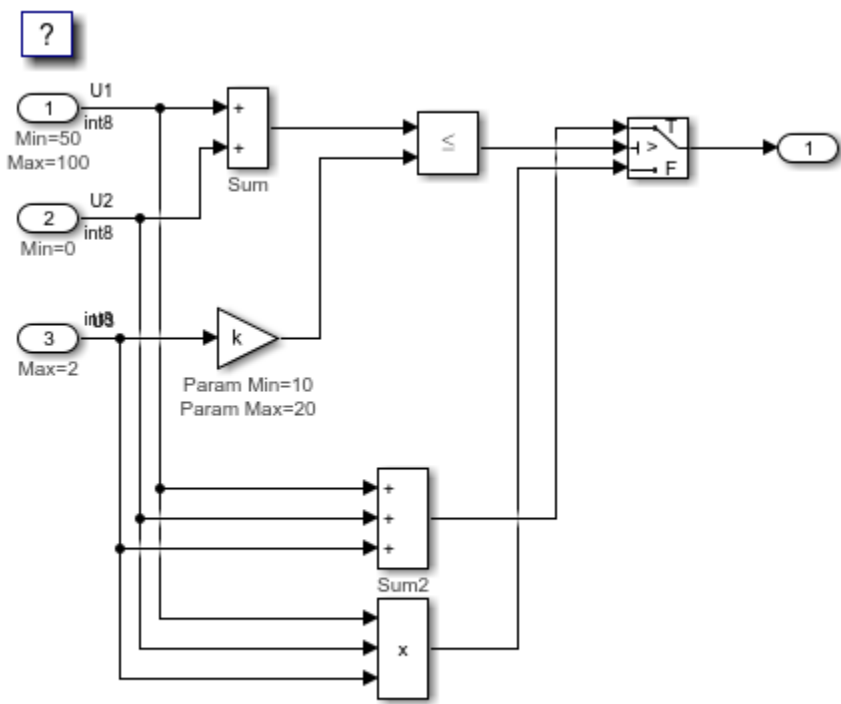
Optimize Generated Code Using Specified Minimum and Maximum Values

This example shows how the minimum and maximum values specified on signals and parameters in a model are used to optimize the generated code.

Review Minimum and Maximum Information

Consider the MinMaxOptimization model. In this model, there are minimum and maximum values specified on Inports and on the gain parameter of the Gain block.

```
model = 'MinMaxOptimization';  
open_system(model);
```



Copyright 2010-2024 The MathWorks, Inc.

Generate Code Without This Optimization

First, generate code for this model without considering the min and max values.

```
slbuild(model)

### Searching for referenced models in model 'MinMaxOptimization'.
### Total of 1 models to build.
### Starting build procedure for: MinMaxOptimization
### Successful completion of build procedure for: MinMaxOptimization
```

Build Summary

Top model targets:

Model	Build Reason	Status
MinMaxOptimization	Information cache folder or artifacts were missing.	Code generated and comp
1 of 1 models built (0 models already up to date)		
Build duration: 0h 0m 13.998s		

A portion of MinMaxOptimization.c is listed below.

```
cfile = fullfile('MinMaxOptimization_ert_rtw','MinMaxOptimization.c');
coder.example.extractLines(cfile,'/* Model step', '/* Model initialize', 1, 0);

/* Model step function */
```

```

void MinMaxOptimization_step(void)
{
    int32_T tmp;

    /* Sum: '<Root>/Sum' incorporates:
     * Inport: '<Root>/U1'
     * Inport: '<Root>/U2'
     * Sum: '<Root>/Sum2'
     * Switch: '<Root>/Switch'
     */
    tmp = U1 + U2;

    /* Switch: '<Root>/Switch' incorporates:
     * Gain: '<Root>/Gain'
     * Inport: '<Root>/U3'
     * RelationalOperator: '<Root>/Relational Operator'
     * Sum: '<Root>/Sum'
     */
    if (tmp <= k * U3) {
        /* Outport: '<Root>/Out1' incorporates:
         * Sum: '<Root>/Sum2'
         */
        rtY.Out1 = tmp + U3;
    } else {
        /* Outport: '<Root>/Out1' incorporates:
         * Inport: '<Root>/U1'
         * Inport: '<Root>/U2'
         * Product: '<Root>/Product'
         */
        rtY.Out1 = U1 * U2 * U3;
    }
}

```

Enable This Optimization

- 1 Open the Configuration Parameters dialog box.
- 2 On the **Optimization** pane, select **Optimize using the specified minimum and maximum values**.

Alternatively, you can enable this optimization by setting the command-line parameter.

```
set_param(model, 'UseSpecifiedMinMax', 'on');
```

Generate Code with This Optimization

In the model, with the specified minimum and maximum values for U1 and U2, the sum of U1 and U2 has a minimum value of 50. Considering the range of U3 and the specified minimum and maximum values for the Gain block parameter, the maximum value of the Gain block's output is 40.

The output of the Relational Operator block remains false, and the output of the Switch block remains the product of the three inputs.

Build the model.

```
slbuild(model)
```

```

### Searching for referenced models in model 'MinMaxOptimization'.
### Total of 1 models to build.

```

```
### Starting build procedure for: MinMaxOptimization
### Successful completion of build procedure for: MinMaxOptimization
```

Build Summary

Top model targets:

Model	Build Reason	Status	Build Duration
MinMaxOptimization	Generated code was out of date.	Code generated and compiled.	0h 0m 12.382s

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.781s
```

View the optimized code from MinMaxOptimization.c.

```
cfile = fullfile('MinMaxOptimization_ert_rtw','MinMaxOptimization.c');
coder.example.extractLines(cfile,'/* Model step', '/* Model initialize', 1, 0);
```

```
/* Model step function */
void MinMaxOptimization_step(void)
{
    /* Output: '<Root>/Out1' incorporates:
    *   Inport: '<Root>/U1'
    *   Inport: '<Root>/U2'
    *   Inport: '<Root>/U3'
    *   Product: '<Root>/Product'
    *   Switch: '<Root>/Switch'
    */
    rtY.Out1 = U1 * U2 * U3;
}
```

Close the model and cleanup.

```
bdclose(model)
```

Tips

- To detect mismatches between model and generated code simulations that arise from the use of this parameter, before running normal, accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) simulations, set **Diagnostics > Data Validity > Simulation range checking** to Warning or Error.
- Provide as much design minimum and maximum information as possible. Specify minimum and maximum values for signals and parameters in the model for:
 - Inport and Outport blocks.
 - Block outputs.
 - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks.
 - Simulink.Signal objects.
- Before generating code, test the minimum and maximum values for signals and parameters. Otherwise, optimization might result in numerical mismatch with simulation. You can simulate your model with simulation range checking enabled. If errors or warnings occur, fix these issues before generating code.

- 1 In the Configuration Parameters dialog box, select **Diagnostics > Data Validity**.
 - 2 On the **Data Validity** pane, under **Signals**, set **Simulation range checking** to warning or error.
- Provide design minimum and maximum information upstream of blocks as close to the inputs of the blocks as possible. If you specify minimum and maximum values for a block output, these values are most likely to affect the outputs of the blocks immediately downstream.
 - This optimization does not take into account minimum and maximum values specified for:
 - Merge block inputs. To work around this, use a `Simulink.Signal` object on the Merge block output and specify the range on this object
 - Bus elements.
 - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Output block.
- Output blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.
- Enabling this optimization improves the ability of the Fixed-Point Designer software to eliminate unnecessary utility functions and saturation code from the generated code.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

Programmatic Use

Parameter: UseSpecifiedMinMax

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Limitations

- This optimization does not take into account minimum and maximum values specified for:
 - Merge block inputs. To work around this, use a `Simulink.Signal` object on the Merge block output and specify the range on this object
 - Bus elements.
 - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Output block.

Output blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.

- If you use the Polyspace Code Prover™ software to verify code generated using this optimization, it might mark code that was previously green as orange. For example, if your model contains a division where the range of the denominator does not include zero, the generated code does not include protection against division by zero. Polyspace Code Prover might mark this code orange because it does not have information about the minimum and maximum values specified for the inputs to the division.

The Polyspace Code Prover software does automatically capture some minimum and maximum values specified in the MATLAB workspace, for example, for `Simulink.Signal` and `Simulink.Parameter` objects. In this example, to provide range information to the Polyspace Code Prover software, use a `Simulink.Signal` object on the input of the division and specify a range that does not include zero.

The Polyspace Code Prover software stores these values in a Data Range Specification (DRS) file. However, they do not capture minimum and maximum values specified in your Simulink model. To provide additional min/max information to Polyspace Code Prover, you can manually define a DRS file. For more information, see the Polyspace Code Prover documentation.

- If you are using double-precision data types and the **Code Generation > Interface > Support non-finite numbers** configuration parameter is selected, this optimization does not occur.
- This optimization does not apply to reusable subsystems.
- The Model Advisor check Check safety-related optimization settings for specified minimum and maximum values generates a warning if this option is selected. For many safety-critical applications, removing dead code automatically is unacceptable because doing so might make code untraceable.
- If you specify maximum and minimum values for signals or parameters in the model that have storage classes other than `Auto`, this optimization does not occur.

Version History

Introduced in R2010b

See Also

Topics

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values

Control generation of protective code that saturates out-of-range tunable parameter expression values

Model Configuration Pane: Code Generation / Optimization

Description

The **Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values** parameter specifies to remove code that saturates the values of tunable parameters expressions.

Settings

on (default) | off

On

Does not generate code that saturates the value of out-of-range tunable parameter expressions to prevent integer overflow. Select this check box if code efficiency is critical to your application.

Off

Generates code that guards against overflow by saturating the value of tunable parameter expressions in between upper and lower limits.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM)
Safety precaution	No recommendation

Programmatic Use

Parameter: EfficientTunableParamExpr

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2019b

See Also

Topics

"Model Configuration Parameters: Code Generation Optimization" on page 10-2

“Code Efficiency”

“Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow”

Remove code that protects against integer division arithmetic exceptions

Control generation of protective code for division arithmetic exceptions

Model Configuration Pane: Code Generation / Optimization

Description

The **Remove code that protects against integer division arithmetic exceptions** parameter specifies whether to remove code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.

Settings

off (default) | on

On

Does not generate code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data. To retain bit-true agreement between simulation results and results from generated code, check that your model does not produce division by zero or `INT_MIN/-1` operations, where the quotient cannot be represented in the data type.

Off

Generates code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution, ROM)
Safety precaution	Off

Programmatic Use

Parameter: NoFixptDivByZeroProtection

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2006a

See Also

Topics

“Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data”

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

“Code Efficiency”

Use signal labels to guide buffer reuse

Reuse buffers by using signal labels

Model Configuration Pane: Code Generation / Optimization

Description

The **Use signal labels to guide buffer reuse** parameter instructs the code generator to use the same signal memory for signals with the same label.

Settings

off (default) | on

On

The code generator uses signal labels as a guide for which buffers to reuse.

Off

The code generator ignores signal labels when implementing buffer reuse.

Tips

If your model has the optimal parameter settings for removing data copies, you might be able to remove additional data copies by using signal labels. After studying the generated code and the Static Code Metrics Report and identifying blocks for whose input and output signals you would like to reuse, you can add labels to signal lines and enable the **Use signal labels to guide buffer reuse** parameter. If possible, the code generator implements the reuse specification.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM, RAM)
Safety precaution	No impact

Programmatic Use

Parameter:LabelGuidedReuse

Type:character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2018a

See Also

Topics

[“Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse”](#)

[“Model Configuration Parameters: Code Generation Optimization”](#)

Operator to represent Bitwise and Logical Operator blocks

Specify operator type in generated code

Model Configuration Pane: Code Generation / Optimization

Description

The **Operator to represent Bitwise and Logical Operator blocks** optimizes the generated code with Logical or Bitwise operators or a combination of both. The supported data type is Boolean.

Settings

Same as modeled (default) | Logical Operator | Bitwise Operator

Same as modeled

Generate code with operators as modeled in Simulink. Generates code with a combination of Bitwise and Logical operators if the model contains both Logical and Bitwise blocks.

Logical Operator

Generate code with Logical operators. Converts Bitwise operator blocks in the model to Logical operators in the generated code.

Bitwise Operator

Generate code with Bitwise operators. Converts Logical operators blocks in the model to Bitwise operators in the generated code. Selecting this option may improve ROM efficiency when the code contains Bitwise operators.

Tips

If you set this parameter to **Bitwise Operator**, the code semantics is not identical to the modeled semantics. Operands that are short-circuited in a logical operation in the model are evaluated in the corresponding bitwise operation in the generated code. Account for this difference in semantics when verifying the generated code.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Bitwise operator (ROM efficiency)
Safety precaution	No impact

Programmatic Use

Parameter: BitwiseOrLogicalOp

Type: character vector

Value: 'Same as modeled' | 'Logical operator' | 'Bitwise operator'
Default: 'Same as modeled'

Version History

Introduced in R2020a

See Also

Level | Specify custom optimizations

Topics

“Model Configuration Parameters: Code Generation Optimization” on page 10-2

“Control Operator Type in Generated Code”

Reuse output buffers of Model blocks

Reuse referenced model buffers if possible

Model Configuration Pane: Code Generation / Optimization

Description

Specify whether to reuse referenced model output buffers whenever possible.

Dependencies

The **Reuse output buffers of Model blocks** optimization has these dependencies:

- The **Signal storage reuse** parameter enables the parameter.
- The optimization is enabled only when the top model and the model referenced by a Model block have the **Reuse output buffers of Model blocks** parameter enabled.

Settings

on (default) | off

On

When you enable the parameter for both the top model and the model referenced by a Model block, the code generator analyzes referenced model contents to determine if buffer reuse is possible. If buffer reuse is possible, it reuses signal memory or generates reusable temporary buffers to hold referenced model outputs, which can significantly reduce RAM consumption. This optimization does not work when function prototype control is applied to the referenced models.

Off

Referenced model outputs are stored in unique locations.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

Programmatic Use

Parameter: ReuseModelBlockBuffer

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2022b

See Also

Signal storage reuse

Topics

“Reduce Memory Usage for Models Containing Referenced Models”

Remove local variable initialization to zero value

Control generation of initialization code for local variables set to zero

Model Configuration Pane: Code Generation / Optimization

Description

The **Remove local variable initialization to zero value** parameter specifies whether to generate initialization code for local variables set to zero. Clear this parameter to generate code that complies with coding standards such as MISRA C++:2008 Rule 0-1-4 (Polyspace Bug Finder).

Settings

on (default) | off

On

Does not generate initialization code for local variables set to zero.

Off

Generates initialization code for local variables set to zero.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	'on'
Safety precaution	No recommendation

Programmatic Use

Parameter: RemoveLocalVariableInitialization

Type: character vector

Values: 'on' | 'off'

Default: 'on'

Limitations

- If you clear this parameter, the code generator does not generate zero initializations for local variables that have these types:
 - struct types
 - Types that contain struct types
 - _m128
 - Types that contain a C++ class, such as `std::string`
 - Types that have symbolic dimensions

- If you clear this parameter, the code generator does not generate zero initializations for local variables that have a continuous state.
- If you clear this parameter, the MAT-file logging parameter does not reflect zero initializations of local variables.

Version History

Introduced in R2023b

See Also

MISRA C++:2008 Rule 0-1-4 (Polyspace Bug Finder)

Topics

“Remove Zero-Initialization Code”

Unify buffer reuse candidates

Analyze buffer reuse candidates to implement efficient reuses

Model Configuration Pane: Code Generation / Optimization

Description

The **Unify buffer reuse candidates** parameter instructs the code generator to first analyze potential buffer reuse candidates and identify the efficient reuses, and then implement them in the generated code. Buffer reuse candidates are buffers that can potentially be reused.

Dependencies

To enable this parameter:

- Select the **Specify custom optimizations** parameter.
- Select the **Signal storage reuse** parameter.

Settings

off (default) | on

On

The code generator first analyzes potential buffer reuse candidates, identifies the efficient reuses, and then implements them in the generated code. Analyzing the potential buffer reuse candidates and identifying the efficient reuses upfront reduces the possibility of missing reuse opportunities, which can reduce RAM consumption. However, for some modeling patterns, this optimization might not eliminate some data copies.

Off

The code generator does not analyze the buffer reuse candidates and does not identify the efficient reuses upfront. Instead, the code generator implements the reuses in phases. Implementing reuse in phases can result in missed reuse opportunities.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

Programmatic Use

Parameter: UnifyBufferReuse

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2025a

See Also

Signal storage reuse

Topics

“Generate Efficient Code Using Unified Analysis”

Use C restrict qualifier

Insert C restrict keyword for function arguments

Model Configuration Pane: Code Generation / Optimization

Description

The **Use C restrict qualifier** parameter instructs the code generator to apply the restrict keyword to function arguments when it might enable additional compiler optimizations.

Dependencies

To enable this parameter:

- Set the **Language standard** parameter to **C99 (ISO)**
- Select the **Specify custom optimizations** parameter.

Settings

off (default) | on

On

Applies the restrict keyword to pointer arguments automatically. The code generator does not apply the restrict keyword if the pointer is aliased.

Off

Does not apply the restrict keyword automatically.

Examples

Apply C Restrict Qualifier in Generated Code

This example shows how to generate code that uses the C restrict keyword. Applying the restrict keyword to pointer arguments can enable additional compiler optimizations that can improve the execution efficiency of the code.

Open And Prepare Example Model

Open the example model `ReusableStorageClass.slx`. The model contains multiple subsystems for which the code generator generates functions that use arguments.

```
model = "ReusableStorageClass";
open_system(model)
```

Configure the model to use the C99 language standard. Open the model configuration parameters and set **Language standard** to **C99 (ISO)**, then click **OK**. Alternatively, set the parameter at the command line.

```
set_param(model, "TargetLangStandard", "C99 (ISO)")
```

Generate Code Without Restrict Keyword

Generate code from the model.

```
set_param(model,"UseRestrictQualifier","off");
slbuild(model);

### Searching for referenced models in model 'ReusableStorageClass'.
### Total of 1 models to build.
### Starting build procedure for: ReusableStorageClass
### Successful completion of build procedure for: ReusableStorageClass
```

Build Summary

Top model targets:

Model	Build Reason	Status
=====		
ReusableStorageClass	Information cache folder or artifacts were missing.	Code generated and co
1 of 1 models built (0 models already up to date)		
Build duration: 0h 1m 19.366s		

The generated functions for the subsystems do not use the restrict keyword.

```
coder.example.extractLines("ReusableStorageClass_ert_rtw/ReusableStorageClass.c", ...
    "static void DeltaX", "static void DeltaSubsystem")

static void DeltaX(const real_T rtu_u1[262144], const real_T rtu_u2[262144],
    real_T rty_z1[261632], real_T rty_z2[261632]);
static void DeltaY(const real_T rtu_u1[261632], const real_T rtu_u2[261632],
    real_T rty_z1[261121], real_T rty_z2[261121]);
static void Downsample(const real_T rtu_u1[1048576], const real_T rtu_u2[1048576],
    real_T rty_z1[262144], real_T rty_z2[262144]);
```

Generate Code With Restrict Keyword

Configure the model to generate code that uses the restrict keyword. In the model configuration parameters, on the Code Generation > Optimization pane, turn on **Specify custom optimizations**. In the **Details** section, turn on **Use C restrict Qualifier** and click **OK**. Alternatively, set the parameters from the command line.

```
set_param(model,"OptimizationCustomize","on");
set_param(model,"UseRestrictQualifier","on");
```

Generate code from the model.

```
slbuild(model);

### Searching for referenced models in model 'ReusableStorageClass'.
### Total of 1 models to build.
### Starting build procedure for: ReusableStorageClass
### Successful completion of build procedure for: ReusableStorageClass
```

Build Summary

Top model targets:

Model	Build Reason	Status	Build Durat

```
=====
ReusableStorageClass  Generated code was out of date.  Code generated and compiled.  0h 0m 22.19s
```

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 24.088s
```

The generated functions for the subsystems use the restrict keyword.

```
coder.example.extractLines("ReusableStorageClass_ert_rtw/ReusableStorageClass.c", ...
    "static void DeltaX", "static void DeltaSubsystem")
```

```
static void DeltaX(const real_T *__restrict rtu_u1, const real_T *__restrict
    rtu_u2, real_T *__restrict rty_z1, real_T *__restrict rty_z2);
static void DeltaY(const real_T *__restrict rtu_u1, const real_T *__restrict
    rtu_u2, real_T *__restrict rty_z1, real_T *__restrict rty_z2);
static void Downsample(const real_T *__restrict rtu_u1, const real_T *__restrict
    rtu_u2, real_T *__restrict rty_z1, real_T *__restrict rty_z2);
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	Off

Programmatic Use

Parameter: UseRestrictQualifier

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2026a

Code Generation Parameters: Comments

Model Configuration Parameters: Comments

The **Code Generation > Comments** category includes parameters for configuring the comments in the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

Code Comments are generated automatically or you can add them to the code.

Code comments have the following uses:

- Enhance the readability and traceability of code
- Convey information among users
- Enhance code search in code base

Code Comments can be classified into Auto generated and Custom comments. Auto generated comments are automatically generated by the software during code generation and the user adds Custom comments.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Comments** pane.

Auto generated comments

Parameter	Description
Include comments	Specify which comments are in generated files.
Simulink block comments	Specify whether to insert Simulink block comments.
Trace to model using	Specify format of comments for Simulink blocks, Stateflow elements and MATLAB function blocks.
Stateflow object comments	Specify whether to insert Stateflow object comments.
MATLAB source code as comments	Specify whether to insert MATLAB source code as comments.
Show eliminated blocks	Specify whether to insert eliminated block's comments.
Verbose comments for 'Model default' storage class	Reduce code size or improve code traceability by controlling the generation of comments.
Operator annotations	Specify whether to include operator annotations for Polyspace in the generated code as comments.

Custom comments

Parameter	Description
Simulink block descriptions	Specify whether to insert descriptions of blocks into generated code as comments.

Parameter	Description
Stateflow object descriptions	Specify whether to insert descriptions of Stateflow objects into generated code as comments.
Simulink data object descriptions	Specify whether to insert descriptions of Simulink data objects into generated code as comments.
Requirements in block comments	Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.
Connected annotations in block comments	Specify whether to include connected annotations of blocks into generated code as comments.
Custom comments (MPT objects only)	Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.
MATLAB user comments	Specify whether to include MATLAB user comments as comments.
Custom comments function	Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects.

The following configuration parameters are under the **Advanced parameters**.

Parameter	Description
Comment style	Specify a multi-line or single-line comment style for generated C or C++ code.
Insert Polyspace comments	Specify whether to insert code comments for Polyspace block annotations.

- The code generation software automatically inserts comments into the generated code for custom blocks. Therefore, you do not need to include block comments in the associated TLC file for a custom block.

Note If you have existing TLC files with manually inserted comments for block descriptions, the code generation process emits these comments instead of the automatically generated comments. Consider removing existing block comments from your TLC files. Manually inserted comments might be poorly formatted in the generated code and code-to-model traceability might not work.

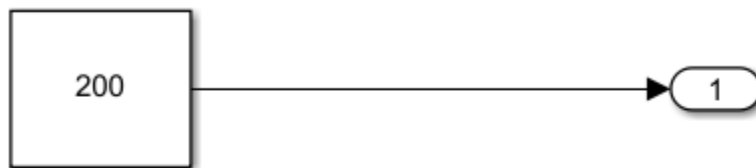
- For virtual blocks or blocks that have been removed due to block reduction, comments are not generated.
- When you configure the code generator to produce code that includes comments, the code generator includes text for model parameters, block names, signal names, and Stateflow object names in the generated code comments. If the text includes characters that are unrepresented in the character set encoding for the model, the code generator replaces the characters with XML escape sequences. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `ア`. For more information, see “Internationalization and Code Generation”.
- When you set the model configuration parameter **Default parameter behavior** to Tunable, the code generator adds different comments about numeric block parameters in the generated code

depending on the numeric value of the block parameter and the output data type. For instance, code generator adds:

- **Computed Parameter** as a comment when the numeric value of the block parameter needs type conversion to match the output data type.
- **Expression** as a comment when the numeric value of the block parameter matches the output data type without type conversion.

Simulink interprets the data type of a numeric parameter as double unless you explicitly specify otherwise. Generate code for the following model:

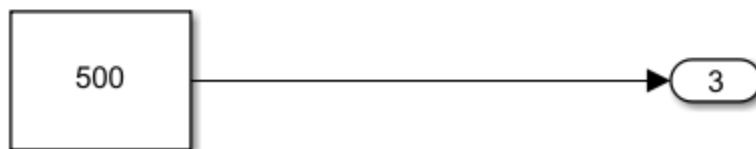
Output data type is double



Output data type is double



output data type is int32



```

// Parameters (auto storage)
struct P_test_parameter_T_ {
    real_T Constant1_Value;           // Expression: 200
                                     // Referenced by: '<Root>/Constant1'

    real_T Constant2_Value;           // Computed Parameter: Constant2_Value
                                     // Referenced by: '<Root>/Constant2'

    int32_T Constant3_Value;          // Computed Parameter: Constant3_Value
                                     // Referenced by: '<Root>/Constant3'
};
  
```

When the constant value is 200 and the output data type is double, the code generator adds **Expression** as a comment. Simulink interprets the data type of the constant value as double and without type conversion it matches the output data type.

When the constant value is uint8(200) and the output data type is double, the code generator adds **Computed Parameter** as a comment. The constant value requires type conversion to match the output data type.

When the constant value is 500 and the output data type is int32, the code generator adds `Computed Parameter` as a comment. The constant value requires type conversion to match the output data type.

See Also

More About

- “Configure Code Comments”
- “Verify Generated Code by Using Code Tracing”

Trace to model using

Specify format of comments

Model Configuration Pane: Code Generation / Comments

Description

Specify format of comments for Simulink blocks, Stateflow elements, and MATLAB function blocks.

Dependencies

- This parameter requires Embedded Coder.
- **Simulink block comments** or **Stateflow object comments** enable this parameter.

Settings

Block path (default) | Simulink identifier

Block path

The generated comment includes the entire block path from the root as the traceability link. For example:

```
/* Outport: '<Root>/Out1' */CustomCodeComments_Y.Out1 = 1;
```

Simulink identifier

The generated comment includes the Simulink identifier number for the corresponding block or object. For example:

```
/* Outport: 'Out1' (':33') */  
CustomCodeComments_Y.Out1 = 1;
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: BlockCommentType

Type: character vector

Value: 'BlockPathComment' (UI Setting Block path) | 'BlockSIDComment' (UI Setting Simulink identifier)

Default: 'BlockPathComment'

Version History

Introduced in R2018a

See Also

Topics

"Model Configuration Parameters: Comments" on page 11-2

Operator annotations

Include operator annotations for Polyspace in generated comments

Model Configuration Pane: Code Generation / Comments

Description

Specify whether to include operator annotations for Polyspace in the generated code as comments.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Include comments** enables this parameter.

Settings

on (default) | off

On

Includes operator annotations in the generated code.

Off

Does not include operator annotations in the generated code.

Tips

- These annotations help document overflow behavior that is due to the way the code generator implements an operation. These operators cannot be traced to an overflow in the design.
- Justify operators that the Polyspace software cannot prove. When this option is enabled, if the code generator uses one of these operators, it adds annotations to the generated code to justify the operators for Polyspace.
- The code generator cannot justify operators that result from the design.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: OperatorAnnotations

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2012b

See Also

Topics

"Model Configuration Parameters: Comments" on page 11-2

"Annotate Code for Justifying Polyspace Checks"

Simulink block descriptions

Insert descriptions of blocks as comments in generated code

Model Configuration Pane: Code Generation / Comments

Description

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Settings

on (default) | off

On

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

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

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: InsertBlockDesc

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Comments" on page 11-2

"Internationalization and Code Generation"

Simulink data object descriptions

Include descriptions of Simulink data objects as comments in generated code

Model Configuration Pane: Code Generation / Comments

Description

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

This parameter applies only to data objects assigned to a built-in storage class or predefined storage class from the `Simulink` package. It does not affect custom storage classes defined in the Embedded Coder Dictionary.

This parameter does not affect `Simulink.LookupTable` or `Simulink.Breakpoint` objects that you configure to appear in the generated code as a structure (for example, by storing all of the table and breakpoint data in a single `Simulink.LookupTable` object).

Dependencies

- This parameter appears for only ERT-based targets.
- This parameter requires an Embedded Coder license when you generate code.

Settings

on (default) | off

On

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

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

Programmatic Use

Parameter: `SimulinkDataObjDesc`

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Comments" on page 11-2

Connected annotations in block comments

Include connected annotations as custom comments in generated code

Model Configuration Pane: Code Generation / Comments

Description

Insert annotations connected to blocks as comments in the generated code.

Annotations not connected to model elements, including area annotations, do not render as comments. Text formatting and images in annotations also do not appear in comments. Annotations converted to markups appear in comments even if hidden. See “Hide Annotations”.

Settings

off (default) | on

On

Include connected annotations as comments in the generated code.

Off

Do not include connected annotations as comments in the generated code.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No Impact
Safety precaution	No Impact

Programmatic Use

Parameter: AnnotationsInComments

Type: character vector

Values: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2024b

See Also

Simulink block descriptions

Topics

“Generate Custom Comments from Block Annotations”

Custom comments (MPT objects only)

Include comments for MPT signal and parameter data in generated code

Model Configuration Pane: Code Generation / Comments

Description

Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code. MPT data objects are objects of the classes `mpt.Parameter` and `mpt.Signal`.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter requires that you include the comments in a function defined in a MATLAB language file or TLC file that you specify with **Custom comments function**.

Settings

off (default) | on

On

Inserts comments just above the identifiers for signal and parameter MPT objects in generated code.

Off

Suppresses the generation of custom comments for signal and parameter identifiers.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

Programmatic Use

Parameter: EnableCustomComments

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Comments” on page 11-2

“Add Custom Comments for Variables in the Generated Code”

Custom comments function

Specify a file that contains comments for MPT signal and parameter data objects

Model Configuration Pane: Code Generation / Comments

Description

Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects. MPT data objects are objects of the classes `mpt.Parameter` and `mpt.Signal`.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Custom comments (MPT objects only)** enables this parameter.

Settings

' ' (default) | string

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

Tip

You might use this option to insert comments that document some or all of the property values of an object.

For an example MATLAB function, open the `CustomCodeComments` model and navigate to the **Custom comments function** configuration parameter:

```
openExample('CustomCodeComments')
```

Recommended Settings

Application	Setting
Debugging	Valid file name
Traceability	Valid file name
Efficiency	Noimpact
Safety precaution	Noimpact

Programmatic Use

Parameter: `CustomCommentsFcn`

Type: character vector

Value: valid file name

Default: ' '

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Comments” on page 11-2

“Add Custom Comments for Variables in the Generated Code”

Stateflow object descriptions

Include descriptions of Stateflow objects in generated comments

Model Configuration Pane: Code Generation / Comments

Description

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Stateflow license.

Settings

on (default) | off

On

Inserts descriptions of Stateflow states, charts, transitions, and graphical functions into generated code as comments. The descriptions come from the **Description** field in Object Properties pane in the Model Explorer for these Stateflow objects. The comments appear just above the code generated for each object.

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

Suppresses the generation of comments for Stateflow objects.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

Programmatic Use

Parameter: SFDataObjDesc

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Comments” on page 11-2

“Internationalization and Code Generation”

Requirements in block comments

Include requirement descriptions in generated comments

Model Configuration Pane: Code Generation / Comments

Description

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires Embedded Coder and Requirements Toolbox™ licenses when generating code.

Settings

off (default) | on

On

Inserts the requirement descriptions that you assign to Simulink blocks into the generated code as comments. The code generator includes the requirement descriptions in the generated code in the following locations.

Model Element	Requirement Description Location
Model	In the main header file <i>model.h</i>
Nonvirtual subsystems	At the call site for the subsystem
Virtual subsystems	At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem does not have a nonvirtual parent, requirement descriptions are located in the main header file for the model, <i>model.h</i> .
Nonsubsystem blocks	In the generated code for the block

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

Off

Suppresses the generation of comments for block requirement descriptions.

Tips

If you use an external *.slmx* file to store your requirement links, to avoid stale comments in generated code, before code generation, you must save any change in your requirement links.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	No recommendation

Programmatic Use

Parameter: ReqsInCode

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Comments” on page 11-2

“How Requirements Information Is Included in Generated Code” (Requirements Toolbox)

MATLAB user comments

Include MATLAB user comments in generated code

Model Configuration Pane: Code Generation / Comments

Description

Specify whether to include MATLAB user comments including both function description comments and other user comments from MATLAB code as comments in the generated code.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Include comments** enables this parameter.

Settings

off (default) | on

On

Inserts MATLAB user comments as comments.

Off

Suppresses comments.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: MATLABFcnDesc

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2010b

See Also

Topics

“Model Configuration Parameters: Comments” on page 11-2

“Include MATLAB Code as Comments in the Generated Code”

Comment style

Specify comment style in generated code

Model Configuration Pane: Code Generation / Comments

Description

Specify comment style in the generated C/C++ code.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

Auto (default) | Multi-line | Single-line

Auto

For C code, generate single- or multiple-line comments delimited by `/*` and `*/`. For C++ code, generate single-line comments preceded by `//`.

Multi-line

Generate single- or multiple-line comments delimited by `/*` and `*/`.

Example of code generated by using the multiline comment style is:

```
/* Sum: '<Root>/Sum' incorporates:
 *   Constant: '<Root>/INC'
 *   UnitDelay: '<Root>/X'
 */
rtDW.X_g++;
```

Single-line

Generate single-line comments preceded by `//`.

Example of code generated by using the single-line comment style is:

```
// Sum: '<Root>/Sum' incorporates:
//   Constant: '<Root>/INC'
//   UnitDelay: '<Root>/X'

rtDW.X_g++;
```

Note For C code generation, select **Single-line** only if your compiler supports it.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: CommentStyle

Type: character vector

Value: Auto | Multi-line | Single-line

Default: Auto

Version History

Introduced in R2014a

See Also

Topics

“Model Configuration Parameters: Comments”

Insert Polyspace comments

Include code comments for Polyspace block annotations

Model Configuration Pane: Code Generation / Comments

Description

Specify whether to insert code comments for Polyspace block annotations.

Dependencies

- This parameter only appears for ERT-based targets.
- **Include comments** enables this parameter.

Settings

off | on

On

Generate code comments for Polyspace block annotations.

Off

Suppresses the generation of comments for Polyspace block annotations.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: InsertPolySpaceComments

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2015b

See Also

Topics

“Model Configuration Parameters: Comments” on page 11-2

Code Generation Parameters: Report

- “Model Configuration Parameters: Code Generation Report” on page 12-2
- “Configure” on page 12-10

Model Configuration Parameters: Code Generation Report

The **Code Generation > Report** category includes parameters for generating and customizing the code generation report. 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 **Code Generation > Report** pane.

Parameter	Description
Create code generation report	Document generated code in an HTML report.
Open report automatically	Specify whether to display code generation reports automatically.
Generate model Web view	Include the model Web view in the code generation report to navigate between the code and model within the same window.
Generate static code metrics	Include static code metrics report in the code generation report.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
Code-to-model	Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.
Model-to-code	Link Simulink blocks, Stateflow objects, and MATLAB functions in a model diagram to corresponding code segments in a generated HTML report so that the generated code for a block can be highlighted on request.
"Configure" on page 12-10	Open the Model-to-code navigation dialog box for specifying a build folder containing previously-generated model code to highlight.
Eliminated / virtual blocks	Include summary of eliminated and virtual blocks in code generation report.
Traceable Simulink blocks	Include summary of Simulink blocks in code generation report.
Traceable Stateflow objects	Include summary of Stateflow objects in code generation report.
Traceable MATLAB functions	Include summary of MATLAB functions in code generation report.
Summarize which blocks triggered code replacements	Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

See Also

More About

- [“Report Generation”](#)
- [“Model Configuration Set Customization”](#)

Generate static code metrics

Generate static code metrics report

Model Configuration Pane: Code Generation / Report

Description

Generate static code metrics, which can be viewed in the Code view or in a static code metrics report.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Settings

off (default) | on

On

Generate static code metrics.

Off

Omit static code metrics.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: GenerateCodeMetricsReport

Type: Boolean

Value: on | off

Default: off

Version History

Introduced in R2011b

See Also

Topics

“Model Configuration Parameters: Code Generation Report”

“Static Code Metrics”

Code-to-model

Include hyperlinks in the code generation report that link to the model diagram

Model Configuration Pane: Code Generation / Report

Description

Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.
- You must select **Include comments** on the **Code Generation > Comments** pane to use this parameter.

Settings

off (default) | on

On

Includes hyperlinks in the code generation report that link code to corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram. The hyperlinks provide traceability for validating generated code against the source model.

Off

Omits hyperlinks from the generated report.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: IncludeHyperlinkInReport

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Code Generation Report"

"HTML Code Generation Report Extensions"

Model-to-code

Enable code highlighting in the code generation report when you click on Simulink blocks, Stateflow objects, and MATLAB functions

Model Configuration Pane: Code Generation / Report

Description

Link Simulink blocks, Stateflow objects, and MATLAB functions in a model diagram to corresponding code segments in a generated HTML report so that the generated code for a block can be highlighted on request.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.
- You must select the following parameters to use this parameter:
 - **Include comments** on the **Code Generation > Comments** pane
 - At least one of the following:
 - **Eliminated / virtual blocks** on page 12-11
 - **Traceable Simulink blocks** on page 12-13
 - **Traceable Stateflow objects** on page 12-15
 - **Traceable MATLAB functions** on page 12-17

Settings

off (default) | on

On

Includes model-to-code highlighting support in the code generation report. To highlight the generated code for a Simulink block, Stateflow object, or MATLAB script in the code generation report, right-click the item. To add the Embedded Coder app options to the menu, point to **Select Apps** and click the Embedded Coder button . Then, in the Embedded Coder app section, select **Navigate to C Code**.

Off

Omits model-to-code highlighting support from the generated report.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: GenerateTraceInfo

Type: Boolean

Value: on | off

Default: off

Version History

Introduced in R2007b

See Also

Topics

“Model Configuration Parameters: Code Generation Report”

“HTML Code Generation Report Extensions”

Configure

Description

Open the Model-to-code navigation dialog box. Through this dialog box you specify a build folder containing previously generated model code to highlight. When you apply your build folder selection, Embedded Coder attempts to load traceability information from the earlier build, if you selected **Model-to-code** on page 12-8 parameter.

Category: Code Generation > Report

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Model-to-code** parameter.

See Also

Related Examples

- “Reload Existing Traceability Information”
- “Model Configuration Parameters: Code Generation Report”

Eliminated / virtual blocks

Include a summary of eliminated and virtual blocks in the code generation report

Model Configuration Pane: Code Generation / Report

Description

Include summary of eliminated and virtual blocks in the **Traceability Report** section of the code generation report.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Settings

off (default) | on

On

Include summary of eliminated and virtual blocks in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of eliminated and virtual blocks.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: GenerateTraceReport

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2008a

See Also

Topics

[“Model Configuration Parameters: Code Generation Report”](#)

[“HTML Code Generation Report Extensions”](#)

[“Customize Traceability Reports”](#)

Traceable Simulink blocks

Include a summary of Simulink blocks and the corresponding code locations in the code generation report

Model Configuration Pane: Code Generation / Report

Description

Includes a summary of Simulink blocks and the corresponding code locations in the **Traceability Report** section of the code generation report.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Settings

off (default) | on

On

Includes a summary of Simulink blocks and the corresponding code locations in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of Simulink blocks.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: GenerateTraceReportSl

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2008a

See Also

Topics

[“Model Configuration Parameters: Code Generation Report”](#)

[“HTML Code Generation Report Extensions”](#)

[“Customize Traceability Reports”](#)

Traceable Stateflow objects

Include a summary of Stateflow objects and the corresponding code locations in the code generation report

Model Configuration Pane: Code Generation / Report

Description

Includes a summary of Stateflow objects and the corresponding code locations in the **Traceability Report** section of the code generation report.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Settings

off (default) | on

On

Includes a summary of Stateflow objects and the corresponding code locations in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of Stateflow objects.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: GenerateTraceReportSf

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2008a

See Also

Topics

[“Model Configuration Parameters: Code Generation Report”](#)

[“HTML Code Generation Report Extensions”](#)

[“Customize Traceability Reports”](#)

Traceable MATLAB functions

Include a summary of MATLAB functions and corresponding code locations in the code generation report

Model Configuration Pane: Code Generation / Report

Description

Includes a summary of MATLAB functions and corresponding code locations in the **Traceability Report** section of the code generation report.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** parameter.

Settings

off (default) | on

On

Includes a summary of MATLAB functions and corresponding code locations in the **Traceability Report** section of the code generation report.

Off

Does not include a summary of MATLAB functions.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: GenerateTraceReportEm1

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2008a

See Also

Topics

[“Model Configuration Parameters: Code Generation Report”](#)

[“HTML Code Generation Report Extensions”](#)

[“Customize Traceability Reports”](#)

Summarize which blocks triggered code replacements

Include a summary of replacement functions used and their associated blocks in the code generation report

Model Configuration Pane: Code Generation / Report

Description

Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.

Settings

off (default) | on

On

Include code replacement report in the code generation report.

Note Selecting this option also generates code replacement trace information for viewing in the **Trace Information** tab of the Code Replacement Viewer. The generated information can help you determine why an expected code replacement did not occur.

Off

Omit code replacement report from the code generation report.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: GenerateCodeReplacementReport

Type: Boolean

Value: on | off

Default: off

Version History

Introduced in R2012a

See Also

Topics

“Model Configuration Parameters: Code Generation Report”

Analyze Code Replacements in the Generated Code

“Verify Code Replacement Library”

Determine Why Code Replacement Functions Were Not Used

Generate model Web view

Include the model Web view in the code generation report

Model Configuration Pane: Code Generation / Report

Description

Include the model Web view in the code generation report to navigate between the code and model within the same window. You can share your model and generated code outside of the MATLAB environment. You must have a Simulink Report Generator license to include a Web view (Simulink Report Generator) of the model in the code generation report.

Dependencies

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.
- To enable traceability between the code and model, select **Code-to-model** and **Model-to-code**.

Settings

off (default) | on

Default: Off

On

Include model Web view in the code generation report.

Off

Omit model Web view in the code generation report.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: GenerateWebview

Type: string

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2012a

See Also

“Model Configuration Parameters: Code Generation Report”

Topics

“Generate Web View of Model in Code Generation Report”

Code Generation Parameters: Custom Code

Model Configuration Parameters: Code Generation Custom Code

The **Code Generation > Custom Code** category includes parameters for inserting custom C code into the generated code. These parameters require a Simulink Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Custom Code** pane.

Parameter	Description
Use the same custom code settings as Simulation Target	Specify whether to use the same custom code settings as those in the Simulation Target > Custom Code pane.
Additional code	Specify custom code to include near the top of the generated model source file.
Include headers	Specify custom code to include near the top of the generated model header file.
Initialize code	Specify custom code to include in the generated model initialize function.
Terminate code	Specify custom code to include in the generated model terminate function.
Include directories	Specify a list of include folders to add to the include path.
Source files	Specify a list of additional source files to compile and link with the generated code.
Libraries	Specify a list of additional libraries to link with the generated code.
Defines	Specify preprocessor macro definitions to be added to the compiler command line.

See Also

More About

- “Model Configuration Set Customization”

Model Configuration Parameters: Code Generation Interface

Model Configuration Parameters: Code Generation Interface

The **Code Generation > Interface** category includes parameters for configuring the interface of the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license. Generating code for deep learning models using NVIDIA CUDA deep neural network library (cuDNN) or TensorRT high performance inference libraries for NVIDIA GPUs requires a GPU Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Interface** pane.

Parameter	Description
Code replacement library	Specify a code replacement library the code generator uses when producing code for a model.
Code replacement libraries	Specify multiple code replacement libraries the code generator use when producing code for a model.
Shared code placement	Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.
Support: floating-point numbers	Specify whether to generate floating-point data and operations.
Support: non-finite numbers	Specify whether to generate non-finite data and operations on non-finite data.
Support: complex numbers	Specify whether to generate complex data and operations.
Support: absolute time	Specify whether to generate and maintain integer counters for absolute and elapsed time values.
Support: continuous time	Specify whether to generate code for blocks that use continuous time. This parameter does not apply when you use Embedded Coder for a component model that is configured with a service interface.
Support: variable-size signals	Specify whether to generate code for models that use variable-size signals.
Code interface packaging (component)	Select the packaging for the generated C or C++ code interface.
Code interface packaging (subcomponent)	Select the packaging for the generated C or C++ code subcomponent interface.
Multi-instance code error diagnostic	Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.
Pass root-level I/O as	Control how root-level model input and output are passed to the reusable <code>model_step</code> function.

Parameter	Description
Implement root-level I/O as	Control how root-level model input and output are passed to the nonreusable subcomponent <i>model_step</i> function.
Remove error status field in real-time model data structure	Specify whether to log or monitor error status.
Preserve MATLAB namespaces in generated code on page 14-24	Specify whether generated code for enumeration in MATLAB namespace includes the namespace in the type definition.
Include model types in model class	Specify whether to generate model type definitions in a model class.
Array layout	Specify layout of array data for code generation as column-major or row-major
External functions compatibility for row-major code generation	Select diagnostic action if Simulink encounters a function that has no specified array layout
Generate C API for: signals	Generate C API data interface code with a signals structure.
Generate C API for: parameters	Generate C API data interface code with parameter tuning structures.
Generate C API for: states	Generate C API data interface code with a states structure.
Generate C API for: root-level I/O	Generate C API data interface code with a root-level I/O structure.
External mode	Generate code for the external mode data interface. When using Embedded Coder for a component model that is configured with a service interface, this parameter is not relevant and, therefore, is not supported.
Transport layer	Specify the transport protocol for communications.
MEX-file arguments	Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.
Static memory allocation	Control memory buffer for external mode communication.
Static memory buffer size	Specify the memory buffer size for external mode communication.
Automatically allocate static memory	Automatically allocate static memory for buffers used in external mode communication.
Maximum duration	Specify, in base rate steps, maximum duration that software must consider when determining size of static memory required for external mode communication.
Log signals to MDF file	Log signal and output data from generated code to MDF file.

Parameter	Description
Target library	Specify the target deep learning library used during code generation. cuDNN or TensorRT requires a GPU Coder license.
Learnables Compression	Specify the compression type, "none" or "bfloat16".
ARM Compute Library version	Specify the version of ARM Compute Library.
ARM Compute Library architecture	Specify the ARM architecture supported in the target hardware.
Deep Learning Data Type	Specify the data type to be used by TensorRT library for deep learning code generation. This parameter requires a GPU Coder license.
Auto tuning	Use auto tuning for cuDNN library. Enabling auto tuning allows the cuDNN library to find the fastest convolution algorithms. This parameter requires a GPU Coder license.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
Support non-inlined S-functions	Specify whether to generate code for non-inlined S-functions.
Maximum word length	Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.
Buffer size of dynamically-sized string (bytes)	Number of bytes of the character buffer generated for dynamic string signals without maximum length.
Multiword type definitions	Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.
Use dynamic memory allocation for model initialization	Control how the generated code allocates memory for model data.
Single output/update function	Specify whether to generate the <i>model_step</i> function.
Terminate function required	Specify whether to generate the <i>model_terminate</i> function.
Combine signal/state structures	Specify whether to combine global block signals and global state data into one data structure in the generated code

Parameter	Description
Generate separate internal data per entry-point function	Generate a model's block signals (block I/O) and discrete states (DWork) acting at the same rate into the same data structure.
MAT-file logging	Specify MAT-file logging. This parameter does not apply when you use Embedded Coder for a component model that is configured with a service interface.
MAT-file variable name modifier	Select the text to add to MAT-file variable names.
Existing shared code	Specify folder that contains existing shared code
Suppress generation of fixed-point word size checks	Control generation of preprocessor directives that perform fixed-point word size checks
Remove disable function	Remove unreachable (dead-code) instances of the <code>disable</code> functions from the generated code for ERT-based systems that include model referencing hierarchies.
Remove reset function	Remove unreachable (dead-code) instances of the <code>reset</code> functions from the generated code for ERT-based systems that include model referencing hierarchies.
LUT object struct order for even spacing specification	Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to even spacing.
LUT object struct order for explicit value specification	Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to explicit value.
Generate destructor	Specify whether to generate a destructor for the C++ model class.
Use dynamic memory allocation for model block instantiation	Specify whether generated code uses the operator <code>new</code> , during model object registration, to instantiate objects for referenced models configured with a C++ class interface.
Ignore custom storage classes	Specify whether to apply or ignore custom storage classes.
Ignore test point signals	Specify allocation of memory buffers for test points.
Implement each data store block as a unique access point	Create unique variables for each read/write operation of a Data Store Memory block.
Generate full file banner	Generate full header banner including time stamp.

The following parameters under the **Advanced parameters** are infrequently used and have no other documentation.

Parameter	Description
GenerateSharedConstants	Control whether the code generator generates code with shared constants and shared functions. Default is on. When turned off, the code generator does not generate shared constants.
InferredTypesCompatibility	For compatibility with legacy code including <code>tmwtypes.h</code> , specify that the code generator creates a preprocessor directive <code>#define __TMWTYPES__</code> inside <code>model.h</code> .
TargetLibSuffix <i>character vector - ''</i>	Control the suffix used for naming a target's dependent libraries (for example, <code>_target.lib</code> or <code>_target.a</code>). If specified, the character vector must include a period (.). (For generated model reference libraries, the library suffix defaults to <code>_rtwlib.lib</code> on Windows systems and <code>_rtwlib.a</code> on UNIX systems.). This parameter does not apply for model builds that use the toolchain approach, see "Library Control Parameters"
TargetPreCompLibLocation <i>character vector - ''</i>	Control the location of precompiled libraries. If you do not set this parameter, the code generator uses the location specified in <code>rtwmakecfg.m</code> .
IsERTTarget	Indicates whether or not the currently selected target is derived from the ERT target.
CPPClassGenCompliant	Indicates whether the target supports the ability to generate and configure C++ class interfaces to model code.
ConcurrentExecutionCompliant	Indicates whether the target supports concurrent execution
UseToolchainInfoCompliant	Indicate a custom target is toolchain-compliant.
ModelStepFunctionPrototypeControlCompliant	Indicates whether the target supports the ability to control the function prototypes of initialize and step functions that are generated for a Simulink model. This parameter does not apply when you use Embedded Coder for a component model that is configured with a service interface.
ParMdlRefBuildCompliant	Indicates if the model is configured for parallel builds when building a model that includes referenced models.

Parameter	Description
CompOptLevelCompliant off, on	Set in SelectCallback for a target to indicate whether the target supports the ability to use the Compiler optimization level parameter to control the compiler optimization level for building generated code. Default is off for custom targets and on for targets provided with the Simulink Coder and Embedded Coder products.
ModelReferenceCompliant character vector - off, on	Set in SelectCallback for a target to indicate whether the target supports model reference.

The following parameters are for MathWorks use only.

Parameter	Description
ExtModeTesting	For MathWorks use only.
ExtModeIntrfLevel	For MathWorks use only.
ExtModeMexFile	For MathWorks use only.

See Also

More About

- “Model Configuration Set Customization”

Support: floating-point numbers

Code generation for floating-point data

Model Configuration Pane: Code Generation / Interface

Description

The **Support: floating-point numbers** parameter specifies whether to generate code for floating-point data and operations on floating-point data.

- The setting of this parameter must be the same for the models in a model hierarchy.
- Selecting this parameter enables the **Support: non-finite numbers** parameter and clearing this parameter disables **Support: non-finite numbers**.
- If you select the **MAT-file logging** parameter, select this parameter and the **Support: non-finite numbers** parameter.

Note The command-line values are reverse of the settings values. The value 'on' in the command line corresponds to the description of “Off” in the settings section. The value 'off' in the command line corresponds to the description of “On” in the settings section.

Settings

On (default) | Off

On

Enables generation of floating-point data and related operations.

Off

Disables generation of floating-point data and operations. Code generator produces pure integer code. If you clear this parameter and the code generator encounters floating-point data or expressions, an error occurs. The error message reports offending blocks and parameters.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (GUI), 'on' (command-line) — for integer only
Safety precaution	No impact

Programmatic Use

Parameter: PurelyIntegerCode

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Code Generation Interface" on page 14-2

Support: complex numbers

Code generation for complex data

Model Configuration Pane: Code Generation / Interface

Description

The **Support: complex numbers** parameter specifies whether to generate complex data and operations.

If you clear this parameter for the top model of a model hierarchy, clear the parameter for referenced models.

Settings

On (default) | Off

On

Enables generation of complex numbers and related operations.

Off

Disables generation of complex numbers and operations. If you clear this parameter and the code generator encounters complex data or expressions, an error occurs. The error message reports offending blocks and parameters.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (for real only)
Safety precaution	No impact

Programmatic Use

Parameter: SupportComplex

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface” on page 14-2

Support: absolute time

Code generation for model that includes blocks that depend on absolute or elapsed time

Model Configuration Pane: Code Generation / Interface

Description

The **Support: absolute time** parameter specifies whether the code generator produces code if a model includes blocks that depend on absolute or elapsed time. For models intended for generating production code for a target platform that does not provide timer services, you should avoid using blocks that depend on time. You can use this parameter to check whether the model includes such blocks by clearing the parameter.

Settings

On (default) | Off

On

Specifies that the model includes blocks that depend on absolute or elapsed time values. Select the parameter if the model includes at least one block that depends on time.

If you select the parameter and the model does not include blocks that depend on time, the code generator does not generate code for time counters.

Off

Returns an error if the model includes blocks that depend on absolute or elapsed time.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	No recommendation

Programmatic Use

Parameter: SupportAbsoluteTime

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Code Generation Interface" on page 14-2

"Timer Representation and Computation"

"Timers in Asynchronous Tasks"

Support: continuous time

Code generation for blocks that use continuous time

Model Configuration Pane: Code Generation / Interface

Description

The **Support: continuous time** parameter specifies whether to generate code for blocks that use continuous time.

- This parameter is not relevant and, therefore, is not supported for component models configured with a service code interface.
- Simulink clears this parameter if you select the **Remove error status field in real-time model data structure** parameter.
- Output values read from `ert_main` for a continuous output port can differ from the corresponding output values in logged data if these conditions exist:
 - You select this parameter and the **Single output/update function**.
 - You customize `ert_main.c` or `ert_main.cpp` to read model output values after each base-rate model step.

The difference occurs because, while logged data captures output at major time steps, output read from `ert_main` after the base-rate model step can capture output at intervening minor time steps. This table lists workarounds that eliminate the discrepancy.

Work Around	Customized <code>ert_main.c</code>	Customized <code>ert_main.cpp</code>
Separate the generated output and update functions (clear parameter Single output/update function), and insert code in <code>ert_main</code> to read model output values reflecting only the major time steps. For example, in <code>ert_main</code> , between the <code>model_output</code> call and the <code>model_update</code> call, read the model External outputs global data structure (defined in <code>model.h</code>).	X	
Select parameter Single output/update function . Insert code in the generated <code>model.c</code> or <code>.cpp</code> file that returns model output values reflecting only major time steps. For example, in the model step function, between the output code and the update code, save the value of the model External outputs global data structure (defined in <code>model.h</code>). Then, restore the value after the update code completes.	X	X
Place a Zero-Order Hold block before the continuous output port.	X	X

Settings

Off (default) | On

On

Enables generation of code for blocks that use continuous time.

Off

Disable generation of code for blocks that use continuous time. If you clear this parameter and the model includes blocks that use continuous time, the code generator returns an error.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

Programmatic Use

Parameter: SupportContinuousTime

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface” on page 14-2

“Time-Based Scheduling and Code Generation”

Support: variable-size signals

Code generation for signals that vary in size

Model Configuration Pane: Code Generation / Interface

Description

The **Support: variable-size signals** parameter specifies whether to generate code for models that use variable-size signals.

When using Embedded Coder for a component model that is configured with a service code interface, this parameter affects only service interfaces that use the Direct Access communication method and the ExportedGlobal, ImportedExtern, or ImportedExternPointer storage class.

Settings

Off (default) | On

On

Enables code generation for models that use variable-size signals.

Off

Disables code generation for models that use variable-size signals. If you clear this parameter and the model uses variable-size signals, the code generator returns an error.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: SupportVariableSizeSignals

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2010a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface” on page 14-2

Pass root-level I/O as

Specify how generated code passes root-level input and output to reusable execution entry-point function

Model Configuration Pane: Code Generation / Interface

Description

The **Pass root-level I/O as** parameter controls how root-level model input and output data is passed to a reusable *model_step*, *model_initialize*, or *model_terminate* entry-point function.

Dependency

To enable this parameter, set the **Code interface packaging** parameter to Reusable function.

Settings

Individual arguments (default) | Structure reference | Part of model data structure

Individual arguments

Passes each root-level model input and output value to the entry-point function as a separate argument.

Structure reference

Packs root-level model input into a struct, packs root-level model output into another struct, and passes the structures to the entry-point function as arguments.

Part of model data structure

Packages root-level model input and output into the real-time model data structure rtModel.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: RootIOFormat

Type: character vector

Value: 'Individual arguments' | 'Structure reference' | 'Part of model data structure'

Default: 'Individual arguments'

Version History

Introduced before R2006a

See Also

`model_step` | `model_initialize` | `model_terminate`

Topics

- "Model Configuration Parameters: Code Generation Interface" on page 14-2
- "Configure Generated C Function Interface for Model Entry-Point Functions"
- "Generate Reentrant Code from Top Models"
- "Generate Subsystem Code as Separate Function and Files"
- "Generate Modular Function Code for Nonvirtual Subsystems"

Implement root-level I/O as

Specify how generated code passes root-level input and output to nonreusable subcomponent execution entry-point function

Model Configuration Pane: Code Generation / Interface

Description

The **Implement root-level I/O as** parameter controls how root-level model input and output data is passed to a nonreusable subcomponent (model reference) *model_step*, *model_initialize*, or *model_terminate* entry-point function.

Dependencies

To enable this parameter, set the **Code interface packaging (subcomponent)** parameter to Nonreusable function.

Settings

Auto (default) | Individual arguments | Global variables

Auto

Same behavior as Individual arguments.

Individual arguments

Passes each root-level model input and output value to the subcomponent entry-point function as a separate argument.

Global variables

Passes each root-level model input and output value to the subcomponent entry-point function using global variables.

You cannot select Global variables if you:

- Customize the function prototype of the subcomponent *model_step* function.
- Assign a storage class other than Auto to root-level I/O ports and signals that directly feed into or propagate out of the subcomponent.
- Configure the root-level model with a service interface or AUTOSAR target and connect root-level I/O ports directly to the subcomponent.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: SubcomponentRootIOFormat

Type: character vector

Values: 'Auto' | 'Individual arguments' | 'Global variables'

Default: 'Auto'

Version History

Introduced in R2025a

See Also

Pass root-level I/O as | Code interface packaging (subcomponent)

Remove error status field in real-time model data structure

Whether to log error status

Model Configuration Pane: Code Generation / Interface

Description

The **Remove error status field in real-time model data structure** parameter specifies whether to log error status data in the real-time model data structure `rtModel`. Select this parameter to reduce memory usage.

If your application contains multiple integrated models, the setting of this parameter must be the same for the models to avoid unexpected application behavior. For example, if you select the parameter for one model but not another, an error status might not get registered by the integrated application.

Selecting this parameter clears the **Support: continuous time** parameter.

Settings

Off (default) | On

On

Omits the error status field from the generated real-time model data structure `rtModel`.

Selecting this parameter can cause the code generator to omit the `rtModel` data structure from the generated code.

Off

Includes an error status field in the generated `rtModel` data structure. You can use available macros to:

- Monitor the field for error message data.
- Set the field with error message data.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

Programmatic Use

Parameter: SuppressErrorStatus

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Code Generation Interface" on page 14-2

"Real-Time Model Data Structure"

Preserve MATLAB namespaces in generated code

Specify whether generated code for enumeration in MATLAB namespace includes the namespace in the type definition

Model Configuration Pane: Code Generation / Interface

Description

The **Preserve MATLAB namespaces in generated code** parameter specifies whether the code generator includes the namespace in the generated type definition of an enumerated type that is derived from a `Simulink.IntEnumType` object and is defined inside a MATLAB namespace. For C++ code generation, the code generator generates the enumeration in a namespace. For C code generation, the code generator prefixes the enumeration with the MATLAB namespace.

This table displays the generated code for enumerated type `Colors` in the MATLAB namespace `MyColors` with the parameter set to `On` and `Off`.

Language	On	Off
C	<pre>typedef enum { black, white, } MyColors_Colors;</pre>	<pre>typedef enum { black, white } Colors;</pre>
C++	<pre>namespace MyColors { enum class Colors : int32_t { black, white }; }</pre>	<pre>enum class Colors : int32_t { black, white };</pre>

Settings

On (default) | Off

On

The generated type definition includes the MATLAB namespace where the enumeration is defined.

Off

The generated type definition does not include the MATLAB namespace where the enumeration is defined.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

Programmatic Use

Parameter: PreserveMATLABNamespacesInGeneratedCode

Type: character vector

Values: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2025a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface” on page 14-2

Include model types in model class

Whether code generator places model type definitions in model class

Model Configuration Pane: Code Generation / Interface

Description

The **Include model types in model class** parameter specifies whether the code generator places model type definitions in a model class.

Settings

On (default) | Off

On

Enables generation of model type definitions within the model class namespace. Model type definitions include:

- Root-level inports and outports
- Block inputs and outputs
- DWork vectors
- Block and constant parameters
- Continuous states
- Real-time model data structure (`rtModel`)

The generated code reduces the MISRA 7-3-1 violations.

A user-defined type, such as a `Simulink.Bus` object or a type defined in a MATLAB Function block or Stateflow chart is still generated in the global namespace.

Off

Disables generation of model type definitions within the global namespace.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

Programmatic Use

Parameter: `IncludeModelTypesInModelClass`

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2020a

See Also

Topics

"Model Configuration Parameters: Code Generation Interface" on page 14-2

Ignore custom storage classes

Whether to apply predefined storage classes

Model Configuration Pane: Code Generation / Interface

Description

The **Ignore custom storage classes** parameter specifies whether to apply storage classes that Simulink defines.

Clear this parameter to enable module packaging features.

Settings

off (default) | on

On

If a storage class is individually specified (for example, by using the Code Mappings Editor or a data object), the code generator treats the storage class as `Model default`. If a storage class is specified in the Code Mappings editor on the **Data Defaults** tab, the storage class is treated as "Default". `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer` are three exceptions that are not affected by this parameter setting.

Off

Applies predefined storage classes as configured. Clear this parameter if the model defines data objects that have predefined storage classes.

Tips

- Clear this parameter before configuring data objects that have predefined storage classes.
- The setting for top-level and referenced models must match.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: IgnoreCustomStorageClasses

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Code Generation Interface"

"Organize Parameter Data into a Structure by Using Struct Storage Class"

Ignore test point signals

Whether to allocate memory buffers for test points

Model Configuration Pane: Code Generation / Interface

Description

The **Ignore test point signals** parameter specifies whether to allocate memory buffers for test points.

Settings

Off (default) | on

On

Ignores test points during code generation. The code generator optimizes memory allocation for buffers used for signals that have test points. Selecting this parameter can facilitate transitions from prototyping to deployment and avoids accidental degradation of generated code due to workflow artifacts.

Off

Allocates separate memory buffers for signals that have test points. Clearing this parameter can result in a loss of code generation optimizations, such as reducing memory usage by storing signals in reusable buffers.

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	No impact

Programmatic Use

Parameter: IgnoreTestpoints

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2008b

See Also

Topics

“Model Configuration Parameters: Code Generation Interface”

"Appearance of Test Points in the Generated Code"

"Configure Signals as Test Points"

"How Generated Code Stores Internal Signal, State, and Parameter Data"

Support non-inlined S-functions

Whether to generate code for noninlined S-functions

Model Configuration Pane: Code Generation / Interface

Description

The **Support non-inlined S-functions** parameter specifies whether to generate code for noninlined S-functions.

The S-function interface requires support for floating-point and non-finite numbers. If you select this parameter, Simulink selects the **Support: floating-point numbers** and **Support: non-finite numbers** parameters. When this parameter is selected, if you clear **Support: floating-point numbers** or **Support: non-finite numbers**, the code generator displays a warning.

Settings

Off (default) | On

On

Enables code generation for noninlined S-functions.

Off

Disables code generation for noninlined S-functions. If you clear this parameter and the model includes a noninlined S-function, the code generator returns an error during the build process.

Tip

- Inlining S-functions is highly advantageous in production code generation, for example, for implementing device drivers. In such cases, clear this option to enforce use of inlined S-functions for code generation.
- Non-inlined S-functions require additional memory and computation resources, and can result in significant performance issues. Consider using an inlined S-function when efficiency is a concern.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	No recommendation

Programmatic Use

Parameter: SupportNonInlinedSFcns

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced before R2006a

See Also

Topics

"Model Configuration Parameters: Code Generation Interface"

"S-Functions and Code Generation"

Use dynamic memory allocation for model initialization

Whether to allocate dynamic memory for initializing model data

Model Configuration Pane: Code Generation / Interface

Description

The **Use dynamic memory allocation for model initialization** parameter specifies whether the code generator allocates dynamic memory for initializing model data.

Dependency

To enable this parameter, set the **Code interface packaging (component)** parameter to Reusable function.

Settings

Off (default) | On

On

Enables generation of a function that allocates memory (using `malloc`) for model data structures dynamically.

Off

Disables generation of a dynamic memory allocation function. The generated code allocates memory for model data structures statically.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: GenerateAllocFcn

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2013b

See Also

`model_step`

Topics

“Model Configuration Parameters: Code Generation Interface”

“Configure Generated C Function Interface for Model Entry-Point Functions”

“Generate Reentrant Code from Top Models”

“Generate Subsystem Code as Separate Function and Files”

“Generate Modular Function Code for Nonvirtual Subsystems”

Use dynamic memory allocation for model block instantiation

Whether to allocate dynamic memory for instantiating objects for reference models configured with C++ class interface

Model Configuration Pane: Code Generation / Interface

Description

The **Use dynamic memory allocation for model block instantiation** parameter specifies whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models that are configured with a C++ class interface.

Dependencies

To enable this parameter, set:

- **Language** to C++
- **Code interface packaging** to C++ class.

Settings

Off (default) | On

On

Enables generation of code that uses dynamic memory allocation to instantiate objects for referenced models configured with a C++ class interface. During instantiation of an object for the top model in a model hierarchy, the generated code uses `new` to instantiate objects for referenced models.

Selecting this parameter frees a parent model from having to maintain information about referenced models beyond its direct children.

- If you select this parameter and an out-of-memory error occurs during the use of `new`, be aware that the code generator might throw a `bad_alloc` exception, per the C++ standard. You must provide code to catch and process the `bad_alloc` exception in case an out-of-memory error occurs for a `new` call during construction of the top model object.
- If you select this parameter and the base model contains a Model block, the build process might generate copy constructor and assignment operator functions in the private section of the model class. Those functions prevent pointer members within the model class from being copied by other code.

Off

Disables generation of code that uses `new` to instantiate referenced model objects.

If you clear this parameter, a parent model maintains information about its referenced models, including its direct and indirect children.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

Programmatic Use

Parameter: UseOperatorNewForModelRefRegistration

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2009b

See Also

Topics

“Model Configuration Parameters: Code Generation Interface”

Terminate function required

Whether to generate terminate entry-point function

Model Configuration Pane: Code Generation / Interface

Description

The **Terminate function required** parameter specifies whether to generate a *model_terminate* entry-point function.

Settings

On (default) | Off

On

Enables generation of a *model_terminate* function. This function contains model termination code and should be called as part of system shutdown.

Off

Disables generation of a *model_terminate* function. Clear this parameter if you designed your application to run indefinitely.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: IncludeMdlTerminateFcn

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

model_terminate

Topics

“Model Configuration Parameters: Code Generation Interface”

Combine signal/state structures

Whether to combine global block signal and state data into one structure

Model Configuration Pane: Code Generation / Interface

Description

The **Combine signal/state structures** parameter specifies whether the code generator combines global block signal and state data into one data structure.

Settings

Off (default) | On

On

Enables generation of global block signal data (block I/O) and global state data (DWork vectors) into one data structure.

Off

Disables generation of global block signal and state data into one data structure. The code generator produces separate data structures for global block signals and global states.

Examples

Combine Global Signal and State Data In One Structure

- 1 For a model that generates the following code:

```
/* Block signals (auto storage) */
typedef struct {
    struct {
        uint_T LogicalOperator:1;
        uint_T UnitDelay1:1;
    } bitsForTID0;
} BlockIO;
/* Block states (auto storage) */
typedef struct {
    struct {
        uint_T UnitDelay_DSTATE:1
        uint_T UnitDelay1_DSTATE:1
    } bitsForTID0;
} D_Work;
```

- 2 If you select **Combine signal/state structures**, the generated code now looks like this:

```
/* Block signals and states (auto storage)
for system */
typedef struct {
    struct {
        uint_T LogicalOperator:1;
        uint_T UnitDelay1:1;
        uint_T UnitDelay_DSTATE:1;
```

```
    uint_T UnitDelay1_DSTATE:1;  
  } bitsForTID0;  
} D_Work;
```

Tips

The benefits to selecting this parameter are:

- Enables tighter memory representation through fewer bitfields, which reduces RAM usage
- Enables better alignment of data structure elements, which reduces RAM usage
- Reduces the number of arguments to reusable subsystem and model block functions, which reduces stack usage
- Better readable data structures with more consistent element sorting

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

Programmatic Use

Parameter: CombineSignalStateStructs

Type: character vector

Value: 'on' | 'off'

Default: off

Version History

Introduced in R2011a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface”

“How Generated Code Stores Internal Signal, State, and Parameter Data”

Implement each data store block as a unique access point

Whether to generate unique variable for each read and write operation on Data Store Memory block

Model Configuration Pane: Code Generation / Interface

Description

The **Implement each data store block as a unique access point** parameter specifies whether the code generator produces a unique variable to hold the value for each Data Store Read and Data Store Write block that operates on a Data Store Memory block.

Settings

Off (default) | On

On

Enables generation of a unique variable for each Data Store Read and Data Store Write block that operates on a Data Store Memory block. The unique variables enhance data coherency.

Off

Disables generation of a unique variable for each read and write operation on a Data Store Memory block. The absence of unique variables diminishes data coherency.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: DSAsUniqueAccess

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2018b

See Also

Topics

“Model Configuration Parameters: Code Generation Interface”

“Improve Data Coherency in Generated Code”

Generate separate internal data per entry-point function

Whether to place variables representing signals and states operated on at same rate in same data structure

Model Configuration Pane: Code Generation / Interface

Description

The **Generate separate internal data per entry-point function** parameter specifies whether to place the variables that represent model signals (block I/O data) and discrete states (DWork data) that are operated on at the same rate into the same data structure. If you configure a model to place the data associated with different rates in different structures, the code generator includes prefixes `FuncInternalData0`, `FunctionInternalData1`, and so on in the structure names.

Dependency

To enable this parameter, select the **Combine signal/state structures** parameter.

Settings

Off (default) | On

On

Enables generation of code for storing global block signal data (block I/O) and global state data (DWork vectors) that are operated on at the same rate in one data structure.

Off

Disables generation of code for storing global block signal data and global state data that are operated on at the same rate in one data structure.

Examples

Place Variables Operated On at Same Rate In Same Data Structure

- 1 For a model that generates this code:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
    real_T RTBS2F;          /* '<Root>/RTBS2F' */
    real_T UDS;             /* '<Root>/UDS' */
    real_T Sum3;            /* '<Root>/Sum3' */
    real_T Sum1;            /* '<Root>/Sum1' */
    real_T UDF_DSTATE;      /* '<Root>/UDF' */
    real_T UDS_DSTATE;      /* '<Root>/UDS' */
    real_T RTBS2F_Buffer0;  /* '<Root>/RTBS2F' */
    real_T MIXEDDSM;        /* '<Root>/DSMM' */
    real_T SLOWDSM;         /* '<Root>/DSMS' */
} DW_demo1_T;
```

- 2 If you select **Generate separate internal data per entry-point function**, the generated code looks like this:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
    real_T RTBS2F_Buffer0;          /* '<Root>/RTBS2F' */
    real_T MIXEDDSM;                /* '<Root>/DSMM' */
} DW_demo1_T;

/* Internal Data Grouped For Same Function, for system '<Root>' */
typedef struct {
    real_T RTBS2F;                  /* '<Root>/RTBS2F' */
    real_T Sum3;                    /* '<Root>/Sum3' */
    real_T UDF_DSTATE;              /* '<Root>/UDF' */
} FuncInternalData0_demo1_T;

/* Internal Data Grouped For Same Function, for system '<Root>' */
typedef struct {
    real_T UDS;                     /* '<Root>/UDS' */
    real_T Sum1;                    /* '<Root>/Sum1' */
    real_T UDS_DSTATE;              /* '<Root>/UDS' */
    real_T SLOWDSM;                 /* '<Root>/DSMS' */
} FuncInternalData1_demo1_T;
```

Tips

Selecting this parameter improves cache performance when deploying a model to a multicore hardware environment that meets these requirements:

- The model has multiple rates and the **Treat each discrete rate as a separate task** parameter is selected.
- The model contains multiple exported functions that run at different rates.

The previous models have separate entry-point functions that different cores can call. A core has its own data cache. Placing data for a single entry-point function in the same core data cache improves execution efficiency because the cache accesses are contiguous rather than spread out over multiple cores.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

Programmatic Use

Parameter: GroupInternalDataByFunction

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2021a

See Also

Topics

"Model Configuration Parameters: Code Generation Interface"

"Multicore Processor Targets"

"Time-Based Scheduling"

Multiword type definitions

Whether to define multiword data types as system- or user-defined types

Model Configuration Pane: Code Generation / Interface

Description

The **Multiword type definitions** parameter specifies whether the code generator defines multiword data types as system-defined or user-defined type definitions.

Settings

System defined (default) | User defined

System defined

Defines multiword data types as system type definitions. If the code generator detects multiword usage, the code generator places multiword type definitions in the file `multiword_types.h`.

User defined

Allows you to control how the code generator handles multiword type definitions. Setting the parameter to this value enables the **Maximum word length** parameter, which allows you to specify a maximum word length, in bits, for which the code generator produces multiword type definitions. The code generator places the multiword type definitions in the file `multiword_types.h`. The default maximum word length is 256. If you select 0, the code generator does not generate multiword type definitions in the file `multiword_types.h`.

The maximum word length for multiword types only determines the type definitions generated and does not impact the efficiency of the generated code. If the maximum word length for multiword types is set to 0 or a value that is too small, an error occurs when the generated code is compiled. This error is caused by the generated code using a type that does not have the required type definition. To resolve the error, increase the maximum word length and regenerate the code. If the maximum word length for multiword types is larger than required, then `multiword_types.h` might contain unused type definitions. Unused type definitions do not consume target resources.

Tips

- Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `multiword_types.h` file during code generation. These updates occur when the new model uses multiword types of length greater than those of the other models. You must then recompile and, depending on your development process, reverify previously generated code. To prevent updates to `multiword_types.h`, determine a maximum word length sufficiently big to cover the needs of all models in the hierarchy. Configure every model in the hierarchy to use that same maximum word length.
- The majority of embedded designs do not need multiword types. By setting maximum word length for multiword types to 0, you can prevent use of multiword variables on the target. If you use multiword variables with a maximum word length that is 0 or smaller than required, you are alerted with an error when the generated code is compiled.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MultiwordTypeDef

Type: character vector

Value: 'System defined' | 'User defined'

Default: 'System defined'

Version History

Introduced in R2020a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface”

Generate destructor

Whether to generate destructor for model configured to use C++ class interface

Model Configuration Pane: Code Generation / Interface

Description

The **Generate destructor** parameter specifies whether to generate a destructor for a C++ class interface.

Dependencies

To enable this parameter, set:

- **Language** to C++
- **Code interface packaging** to C++ class

Settings

On (default) | Off

On

Enables generation of a destructor for the C++ model class.

Off

Disables generation of a destructor for the C++ model class.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: GenerateDestructor

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2008b

See Also

Topics

"Model Configuration Parameters: Code Generation Interface"

"Interactively Configure C++ Interface"

MAT-file variable name modifier

Prefix or suffix to add to MAT-file variable names

Model Configuration Pane: Code Generation / Interface

Description

The **MAT-file variable name modifier** parameter specifies the prefix or suffix to add to MAT-file variable names.

Dependency

To enable this parameter, select the **MAT-file logging** parameter.

Settings

rt_ | _rt | none

rt_

Adds prefix text to MAT-file variable names.

_rt

Adds suffix text to MAT-file variable names.

none

Does not add text to MAT-file variable names.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: LogVarNameModifier

Type: character vector

Value: 'none' | 'rt_' | '_rt'

Default: 'rt_'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface”

“Log Program Execution Results”

“Log Data for Analysis”

Existing shared code

Folder that contains shared code to use instead of generated shared utility code

Model Configuration Pane: Code Generation / Interface

Description

The **Existing shared code** parameter specifies a folder that contains existing shared code that the code generator uses during the build process instead of generated shared utility code.

Settings

none (default) | *path to shared code folder*

none

Build process uses locally generated shared utility code.

path to shared code folder

Build process uses shared code (see “Generate Shared Utility Code”) that is available at the path location specified as a character vector. Specify the absolute path or a path relative to the Simulink preference **Code generation folder** (CodeGenFolder).

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ExistingSharedCode

Type: character vector

Value: valid MATLAB variable name

Default: none

Version History

Introduced in R2016b

See Also

sharedCodeUpdate

Topics

“Cross-Release Shared Utility Code Reuse”

“Cross-Release Code Integration”

Remove disable function

Removal of unreachable instances of disable functions from generated code

Model Configuration Pane: Code Generation / Interface

Description

The **Remove disable function** removes unreachable (dead-code) instances of the `disable` functions from the generated code for systems that include model referencing hierarchies. The `disable` functions are generated for models containing blocks with states. If you enable this parameter, the code generator checks whether live code will be removed and issues errors if it finds such code.

For instance, the code generator issues an error to turn off the **Remove disable function** parameter if:

- You enable this parameter for a model containing blocks with states.
- The model is part of a conditionally executed subsystem that disables states.

The error is issued because the generated code needs to call the `disable` functions.

Settings

Off (default) | On

On

Remove unreachable instances of `disable` functions from the generated code for systems that include model referencing hierarchies.

Off

Generate code without removing unreachable instances of the `disable` function.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On
Safety precaution	No recommendation

Programmatic Use

Parameter: RemoveDisableFunc

Type: character vector

Value: valid MATLAB variable name

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2016b

See Also

Remove reset function

Topics

"Remove Reset and Disable Functions from the Generated Code"

Remove reset function

Removal of unreachable instances of reset functions from generated code

Model Configuration Pane: Code Generation / Interface

Description

The **Remove reset function** removes unreachable (dead-code) instances of the `reset` functions from the generated code for systems that include model referencing hierarchies. The `reset` functions are generated for models containing blocks with states. If you enable this parameter, the code generator checks whether live code will be removed and issues errors if it finds such code.

For instance, the code generator issues an error to turn off the **Remove reset function** parameter if:

- You enable this parameter for a model containing blocks with states.
- The model is part of a conditionally executed subsystem that resets states.

The error is issued because the generated code needs to call the `reset` functions to reset states.

Settings

On (default) | Off

On

Remove unreachable instances of `reset` functions from the generated code for systems that include model referencing hierarchies.

Off

Generate code without removing unreachable instances of the `reset` function.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On
Safety precaution	No recommendation

Programmatic Use

Parameter: RemoveResetFunc

Type: character vector

Value: valid MATLAB variable name

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced in R2016b

See Also

Remove disable function

Topics

“Remove Reset and Disable Functions from the Generated Code”

Code replacement libraries

Libraries that define code optimizations for specific target environment

Model Configuration Pane: Code Generation / Interface

Description

The **Code replacement libraries** parameter specifies code replacement libraries that the code generator uses when producing code for a model. A code replacement library defines code optimizations for a specific target environment.

To select a code replacement library, click the **Select** button and use the **Select code replacement libraries-prioritized** dialog box.

Settings

None | *named code replacement libraries*

None

Does not use a code replacement library.

Named code replacement libraries

Generates calls to multiple platform-specific, compiler, custom, or standard code replacement libraries. The list of named libraries depends on:

- Installed support packages.
- System target file, language, language standard, and device vendor configuration.
- Whether you created and registered code replacement libraries.

Tips

- Order the code replacement libraries based on the priority of code replacement. Libraries at the top of the list have a higher priority of replacement.
- Specify multiple code replacement libraries enclosed by single quotes and separated by commas.
- Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.
- If you set the **Shared code placement** parameter to **Shared location** or you generate code for models in a model reference hierarchy:
 - For models that are sharing the location or are in the model hierarchy, you must specify the same code replacement library (same name, tables, and table entries).
 - If you change the name or contents of the code replacement library and rebuild the model from the same folder as the previous build, the code generator reports a checksum warning (see “Manage the Shared Utility Code Checksum”). The warning prompts you to remove the existing folder and stop or stop code generation.
- If both of the following conditions exist for a model that contains Stateflow charts, the Simulink software regenerates code for the charts and recompiles the generated code.

- You *do not* set parameter **Shared code placement** to Shared location.
- Before regenerating code, you change the code replacement library name or contents.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid libraries
Safety precaution	No impact

Programmatic Use

Parameter: CodeReplacementLibrary

Type: character vector

Value: 'None' | 'GNU C99 extensions' | 'AUTOSAR 4.x' | 'GCC ARM Cortex-A' |
'myCustomLibrary'

Default: 'None'

Version History

Introduced in R2012a

See Also

Topics

“Model Configuration Parameters: Code Generation Interface” on page 14-2

“What Is Code Replacement Customization?”

“Develop a Code Replacement Library”

“Optimize Generated Code by Using Multiple Code Replacement Libraries”

Code Generation Parameters: Identifiers

Model Configuration Parameters: Code Generation Identifiers

The **Code Generation > Identifiers** category includes parameters for configuring the identifiers in 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 **Code Generation > Identifiers** pane.

Parameter	Description
Global variables	Customize generated global variable identifiers.
Global types	Customize generated global type identifiers.
Field name of global types	Customize generated field names of global types.
Subsystem methods	Customize generated function names for subsystems and Simulink functions.
Subsystem method arguments	Customize generated function argument names for subsystems and Simulink functions.
Local temporary variables	Customize generated local temporary variable identifiers.
Local block output variables	Customize generated local block output variable identifiers.
Constant macros	Customize generated constant macro identifiers.
Shared utilities identifier format	Customize shared utility identifiers.
Minimum mangle length	Specify the minimum number of characters for generating name-mangling text to help avoid name collisions.
Maximum identifier length	Specify maximum number of characters in generated function, type definition, variable names.
System-generated identifiers	Specify whether the code generator uses shorter, more consistent names for the \$N token in system-generated identifiers.
Generate scalar inlined parameters as	Control expression of scalar inlined parameter values in the generated code.
Use the same reserved names as Simulation Target	Specify whether to use the same reserved names as those specified in the Simulation Target pane.
Reserved names	Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.
Header guard prefix	Specify custom text to prepend to header guards in the generated code.

The following configuration parameters are under the **Advanced parameters**.

Parameter	Description
Shared checksum length	Specify character length of \$C token.
EMX array utility functions identifier format	Customize generated identifiers for emxArray (embeddable mxArray) utility functions.
EMX array types identifier format	Customize generated identifiers for emxArray (embeddable mxArray) types.
Custom token text	Specify text to insert for \$U token.
Duplicate enumeration member names	Select the diagnostic action to take if the code generator detects two enumeration types with same member names. This parameter applies to only enumeration with imported data scope and the same storage type and value.
Signal naming	Specify rules for naming signals in generated code.
MATLAB function	
Parameter naming	Specify rule for naming parameters in generated code.
MATLAB function	
#define naming	Specify rule for naming #define parameters (defined with storage class Define (Custom)) in generated code.
MATLAB function	

See Also

More About

- “Model Configuration Set Customization”

Global variables

Customize generated global variable identifiers

Model Configuration Pane: Code Generation / Identifiers

Description

Customize automatically-generated global variable identifiers. This parameter does not affect global variables that the code generator generates for storage classes and services.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

\$R\$N\$M (default) | string

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 the following format tokens.

Token	Description
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
\$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
\$U	Insert text that you specify for the \$U token. Use the Custom token text parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.

- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This parameter setting only determines the name of objects, such as signals and parameters, if the object is set to **Auto**.
- For referenced models, if the **Global variables** parameter does not contain a \$R token (which represents the name of the reference model), code generation prepends the \$R token to the identifier format.

You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

- 1 On the **Modeling** tab, select **Model Advisor**.
- 2 Select **By Task**.
- 3 Run the **Check code generation identifier formats used for model reference** check.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrGlobalVar

Type: character vector

Value: valid combination of tokens

Default: \$R\$N\$M

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

Global types

Customize generated global type identifiers

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated global type identifiers.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

`$N$R$M_T` (default) | string

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 the following format tokens.

Token	Description
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
\$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
\$U	Insert text that you specify for the \$U token. Use the Custom token text parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.

- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For referenced models, if the **Global types** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

- 1 On the **Modeling** tab, select **Model Advisor**.
- 2 Select **By Task**.
- 3 Run the **Check code generation identifier formats used for model reference** check.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: `CustomSymbolStrType`

Type: character vector

Value: valid combination of tokens

Default: `$N$R$M_T`

Version History

Introduced in R2006b

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

Field name of global types

Customize generated field names of global types

Model Configuration Pane: Global Variables / Identifiers

Description

Customize generated field names of global types.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

`$N$M` (default) | string

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 the following format tokens.

Token	Description
\$A	Insert data type acronym into signal and work vector identifiers. For example, <code>i32</code> for <code>int32_t</code> .
\$H	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the text <code>root_</code> . For blocks at the subsystem level, the tag is of the form <code>sN_</code> , where <code>N</code> is a unique system number assigned by the Simulink software.
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
\$U	Insert text that you specify for the <code>\$U</code> token. Use the Custom token text parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.

- The **Maximum identifier length** setting does not apply to type definitions.
- This option does not impact objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrField

Type: character vector

Value: valid combination of tokens

Default: \$N\$M

Version History

Introduced in R2006b

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

Subsystem methods

Customize generated function names for subsystems and Simulink functions

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated function names for subsystems and Simulink functions.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

`$R$N$M$F` (default) | string

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 the following format tokens.

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

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, typedef statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify \$R, the code generator includes the model name in the typedef.
- This option does not impact objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).
- For referenced models, if the **Subsystem methods** parameter does not contain a \$R token (which represents the name of the reference model), code generation prepends the \$R token to the identifier format.

You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

- 1 On the **Modeling** tab, select **Model Advisor**.
- 2 Select **By Task**.
- 3 Run the **Check code generation identifier formats used for model reference** check.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrFcn

Type: character vector

Value: valid combination of tokens

Default: \$R\$N\$M\$F

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

Subsystem method arguments

Customize generated function argument names for subsystems and Simulink functions

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated function argument names for subsystems and Simulink functions.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

`rt$I$N$M` | string

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated argument name. The macro can include a combination of the following format tokens.

Token	Description
\$I	• Insert u if the argument is an input. • Insert y if the argument is an output. • Insert uy if the argument is an input and output. Optional.
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. Recommended to maximize readability of generated code.
\$U	Insert text that you specify for the \$U token. Use the Custom token text parameter to specify this text.

Tips

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

- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrFcnArg

Type: character vector

Value: valid combination of tokens

Default: rt\$I\$N\$M

Version History

Introduced in R2009b

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

Local temporary variables

Customize generated local temporary variable identifiers

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated local temporary variable identifiers.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

\$N\$M (default) | string

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 the following format tokens.

Token	Description
\$A	Insert data type acronym (for example, i32 for integers) into signal and work vector identifiers.
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated.
\$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. Use the Custom token text parameter to specify this text.

Tips

- Avoid name collisions. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See "Control Case of Identifiers by Using Basic Token Decorators".

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrTmpVar

Type: character vector

Value: valid combination of tokens

Default: \$N\$M

Version History

Introduced in R2006b

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

Local block output variables

Customize generated local block output variable identifiers

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated local block output variable identifiers.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

`rtb_$$M (default) | string`

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 the following format tokens.

Token	Description
\$A	Insert data type acronym (for example, <code>i32</code> for integers) into signal and work vector identifiers.
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
\$U	Insert text that you specify for the \$U token. Use the Custom token text parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrBlkIO

Type: character vector

Value: valid combination of tokens

Default: rtb_ \$N\$M

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

Constant macros

Customize generated constant macro identifiers

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated constant macro identifiers. Constant macros appear in the generated code for parameters that use the storage class Auto when you set **Configuration Parameters > Optimization > Default parameter behavior** to Inlined and **Configuration Parameters > Identifiers > Generate scalar inlined parameters as** to Macros.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

\$R\$N\$M (default) | string

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 the following format tokens.

Token	Description
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
\$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing. This token is ignored when generating identifiers for constant parameters.
\$U	Insert text that you specify for the \$U token. Use the Custom token text parameter to specify this text.

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

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- For non-constant parameters in referenced models, if the **Constant macros** parameter does not contain a \$R token (which represents the name of the reference model), code generation prepends the \$R token to the identifier format.

You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

- 1 On the **Modeling** tab, select **Model Advisor**.
- 2 Select **By Task**.
- 3 Run the **Check code generation identifier formats used for model reference** check.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrMacro

Type: character vector

Value: valid combination of tokens

Default: \$R\$N\$M

Version History

Introduced in R2006b

See Also

Default parameter behavior | Generate scalar inlined parameters as

Topics

"Model Configuration Parameters: Code Generation Identifiers" on page 15-2

"Identifier Format Control"

"Avoid Identifier Naming Collisions"

Shared utilities identifier format

Customize shared utility identifiers

Model Configuration Pane: Code Generation / Identifiers

Description

Customize shared utility identifiers.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Appears only for C++ and AUTOSAR code generation.
- Requires Embedded Coder when generating code.

Settings

`$N$C` (default) | string

Customize generated shared utility identifier names.

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 the following format tokens.

Token	Description
<code>\$N</code>	Insert system-generated name of the object for which the shared utility identifier is generated. Optional.
<code>\$C</code>	Insert eight-character conditional checksum when <code>\$N</code> is not specified or the Maximum identifier length does not accommodate the full length of <code>\$N</code> . Modify checksum character length by using Shared checksum length parameter. Required.
<code>\$R</code>	Insert root model name into identifier, replacing unsupported characters with the underscore (<code>_</code>) character.
<code>\$U</code>	Insert text that you specify for the <code>\$U</code> token. Use the Custom token text parameter to specify this text.

Tips

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate.
- The checksum token `$C` is required. If `$C` is specified without `$N` or `$R`, the checksum is included in the identifier name. Otherwise, the code generator includes the checksum when necessary to prevent name collisions.

- To control the case (upper or lower case) of the text that each token represents, include decorators such as [U_] in your macro. See “Control Case of Identifiers by Using Basic Token Decorators”.
- If you specify \$N or \$R, then the checksum is included in the identifier name only when the identifier length is too short to accommodate the fully expanded format text. The code generator includes the checksum and truncates \$N or \$R until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.
- If you specify \$N and \$R, then the checksum is included in the identifier name only when the identifier length is too short to accommodate the fully expanded format text. The code generator includes the checksum and truncates \$N, and if required, \$R, until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.
- Descriptive text helps make the identifier name more accessible.
- For versions prior to R2016a, the **Shared utilities identifier format** parameter does not support the \$R token. For a model, if the **Shared utilities identifier format** parameter includes a \$R token, and you export the model to a version prior to R2016a, the **Shared utilities identifier format** parameter defaults to \$N\$C.
- For C code generation, use the Embedded Coder dictionary to create a function customization template that specifies the naming rule, then apply the template by using the Code Mappings editor.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrUtil

Type: character vector

Value: valid combination of tokens

Default: \$N\$C

Version History

Introduced in R2013b

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Identifier Format Control”

“Generate Shared Utility Code”

Minimum mangle length

Specify the minimum number of characters for name-mangling

Model Configuration Pane: Code Generation / Identifiers

Description

Increase the minimum number of characters for generating name-mangling text to help avoid name collisions.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

1 (default) | integer

Minimum: 1

Maximum: 15

Specify an integer value that indicates the minimum number of characters the code generator uses when generating name-mangling text. The maximum possible value is 15. The minimum value automatically increases during code generation as a function of the number of collisions. A larger value reduces the chance of identifier disturbance when you modify the model.

Tips

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	1
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: MangleLength

Type: integer

Value: value between 1 and 15

Default: 1

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Avoid Identifier Naming Collisions”

System-generated identifiers

Specify whether to use shorter names for the \$N token in system-generated identifiers

Model Configuration Pane: Code Generation / Identifiers

Description

Specify whether the code generator uses shorter, more consistent names for the \$N token in system-generated identifiers.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

Shortened (default) | Classic

Classic

Generate longer identifier names for the \$N token. For example, for a model named `sym`, if:

- Global variables is `$N$R$M`, the block state identifier is `DWork_sym`.
- Global types is `$R$N$M`, the block state type is a structure named `sym_D_Work`.

Shortened

Shorten identifier names for the \$N token to allow more space for user names. This option provides a more predictable and consistent naming system that uses camel case, no underscores or plurals, and consistent abbreviations for both a type and a variable. For example, for a model named `sym`, if:

- Global variables is `$N$R$M`, the block state identifier is `DW_sym`.
- Global types is `$R$N$M`, the block state type is a structure named `sym_DW`.

System-generated identifiers per model

Classic	Shortened	Data Representation	Description
BlockIO, B	B	Type, Global Variable	Block signals of the system
ExternalInputs	ExtU	Type	Block input data for root system
ExternalInputSizes	ExtUSize	Type	Size of block input data for the root system (used when inputs are variable dimensions)
ExternalOutputs	ExtY	Type	Block output data for the root system
ExternalOutputSizes	ExtYSize	Type	Size of block output data for the root system
U	U	Global Variable	Input data
USize	USize	Global Variable	Size of input data
Y	Y	Global Variable	Output data
YSize	YSize	Global Variable	Size of output data
Parameters	P	Type, Global Variable	Parameters for the system
ConstBlockIO	ConstB	Const Type, Global Variable	Block inputs and outputs that are constants
MachineLocalData, Machine	MachLocal	Const Type, Global Variable	Used by S-function targets
ConstParam, ConstP	ConstP	Const Type, Global Variable	Constant parameters in the system
ConstParamWithInit, ConstWithInitP	ConstInitP	Const Type, Global Variable	Initialization data for constant parameters in the system
D_Work, DWork	DW	Type, Global Variable	Block states in the system
MassMatrixGlobal	MassMatrix	Type, Global Variable	Used for physical modeling blocks
PrevZCSigStates, PrevZCSigState	PrevZCX	Type, Global Variable	Previous zero-crossing signal state
ContinuousStates, X	X	Type, Global Variable	Continuous states
StateDisabled, Xdis	XDis	Type, Global Variable	Status of an enabled subsystem
StateDerivatives, Xdot	XDot	Type, Global Variable	Derivatives of continuous states at each time step
ZCSignalValues, ZCSignalValue	ZCV	Type, Global Variable	Zero-crossing signals
DefaultParameters	DefaultP	Global Variable	Default parameters in the system

Classic	Shortened	Data Representation	Description
GlobalTID	GlobalTID	Global Variable	Used for sample time for states in referenced models
InvariantSignals	Invariant	Global Variable	Invariant signals
NSTAGES	NSTAGES	Global Variable	Solver macro
Object	Obj	Global Variable	Used by ERT C++ code generation to refer to referenced model objects
TimingBridge	TimingBrdg	Global Variable	Timing information stored in different data structures
SharedDSM	SharedDSM	Type, Global Variable	Shared local data stores, which are Data Store Memory blocks with Share across model instances selected
InstP	InstP	Type, Global Variable	Parameter arguments for the system

System-generated identifier names per referenced model or reusable subsystem

Classic	Shortened	Data Representation	Description
rtB, B	B	Type, Global Variable	Block signals of the system
rtC, C	ConstB	Type, Global Variable	Block inputs and outputs that are constants
rtDW, DW	DW	Type, Global Variable	Block states in the system
rtMdlrefDWork, MdlrefDWork	MdlRefDW	Type, Global Variable	Block states in referenced model
rtP, P	P	Type, Global Variable	Parameters for the system
rtRTM, RTM	RTM	Type, Global Variable	RT_Model structure
rtX, X	X	Type, Global Variable	Continuous states in model reference
rtXdis, Xdis	XDis	Type, Global Variable	Status of an enabled subsystem
rtXdot, Xdot	XDot	Type, Global Variable	Derivatives of the S-function's continuous states at each time step
rtZCE, ZCE	ZCE	Type, Global Variable	Zero-crossing enabled
rtZCSV, ZCSV	ZCV	Type, Global Variable	Zero-crossing signal values
rtSharedDSM, SharedDSM	SharedDSM	Type, Global Variable	Shared local data stores, which are Data Store Memory blocks with Share across model instances selected
rtInstP, InstP	InstP	Type, Global Variable	Parameter arguments for the system

Note When you set the **System-generated identifiers** parameter to **Shortened**, while generating an identifier for a global variable of a referenced model which is a run-time parameter, the code generator adds a prefix `rtP_` to the variable name for the **\$N** token.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: InternalIdentifier

Type: character vector
Value: Classic | Shortened
Default: Shortened

Version History

Introduced in R2013a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2
“Construction of Generated Identifiers”
“Avoid Identifier Naming Collisions”
“Data Structures in the Generated Code”
“Identifier Format Control”
“Reserved Keywords”

Generate scalar inlined parameters as

Express scalar inlined parameter values in the generated code as literals or macros

Model Configuration Pane: Code Generation / Identifiers

Description

Control expression of scalar inlined parameter values in the generated code. Block parameters appear inlined in the generated code when you set **Configuration Parameters > Optimization > Default parameter behavior** to Inlined.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

Literals (default) | Macros

Literals

Generates scalar inlined parameters as numeric constants.

Macros

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

Examples

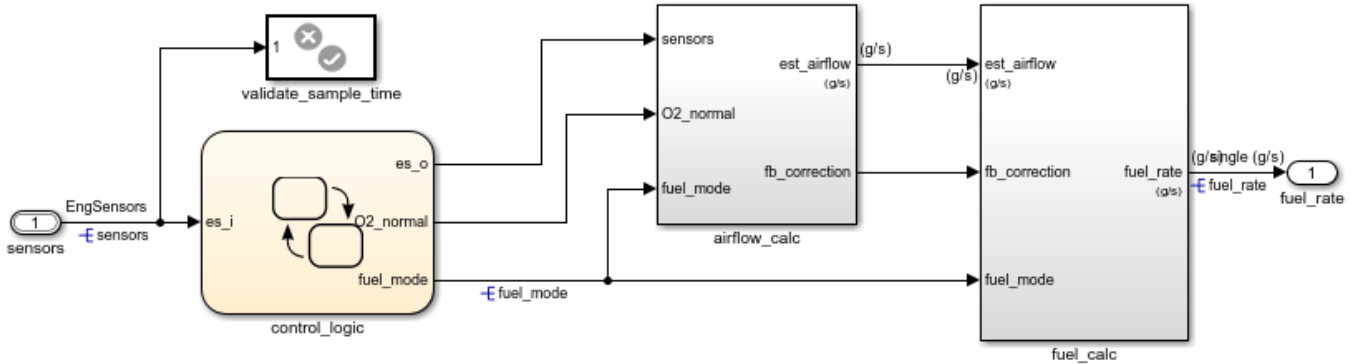
Improve Code Readability by Generating Block Parameter Values as Macros

When you generate efficient code by inlining the numeric values of block parameters (with the configuration parameter **Default parameter behavior**), you can configure scalar parameters to appear as macros instead of literal numbers. Each macro has a unique name that is based on the name of the corresponding block parameter.

Open the example model `sldemo_fuelsys_dd_controller`.

`sldemo_fuelsys_dd_controller`

Fuel Rate Controller



Copyright 1990-2022 The MathWorks, Inc.

The model uses these configuration parameter settings:

- **Default parameter behavior** set to Inlined.
- **System target file** set to ert.tlc.

Set the configuration parameter **Generate scalar inlined parameters as** to Macros.

```
set_param('sldemo_fuelsys_dd_controller', 'InlinedPrmAccess', 'Macros')
```

Generate code from the model.

```
slbuild('sldemo_fuelsys_dd_controller')
```

```
### Searching for referenced models in model 'sldemo_fuelsys_dd_controller'.
### Total of 1 models to build.
### Starting build procedure for: sldemo_fuelsys_dd_controller
### Successful completion of code generation for: sldemo_fuelsys_dd_controller
```

Build Summary

Top model targets:

Model	Build Reason	Status
sldemo_fuelsys_dd_controller	Information cache folder or artifacts were missing.	Code generated

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 31.608s

The header file `sldemo_fuelsys_dd_controller_private.h` defines several macros that represent inlined (nontunable) block parameters. For example, the macros `rtCP_DiscreteFilter_NumCoe_EL_0` and `rtCP_DiscreteFilter_NumCoe_EL_1` represent floating-point constants.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw', ...
    'sldemo_fuelsys_dd_controller_private.h');
```

```
coder.example.extractLines(file, '#define rtCP_DiscreteFilter_NumCoe_EL_0', ...
    'rtCP_DiscreteFilter_NumCoe_EL_1', 1, 1)
```

```
#define rtCP_DiscreteFilter_NumCoe_EL_0 (8.7696F)
#define rtCP_DiscreteFilter_NumCoe_EL_1 (-8.5104F)
```

The comments above the macro definitions indicate that the code generated for a Discrete Filter block uses the macros.

```
coder.example.extractLines(file, 'Computed Parameter: DiscreteFilter_NumCoe', ...
    'Referenced by: '<S12>/Discrete Filter'', 1, 1)
```

```
/* Computed Parameter: DiscreteFilter_NumCoe
 * Referenced by: '<S12>/Discrete Filter'
```

Click the hyperlink to navigate to the block in the model.

Tips

- If you configure a block parameter to use the storage class **Define**, the generated code uses variables with **#define** macros to represent the parameter, and the storage class setting overrides **Generate scalar inlined parameter as**.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Macros
Efficiency	Literals
Safety precaution	No impact

Programmatic Use

Parameter: InlinedPrmAccess

Type: character vector

Value: Literals | Macros

Default: Literals

Version History

Introduced before R2006a

See Also

Default parameter behavior

Topics

"Model Configuration Parameters: Code Generation Identifiers" on page 15-2

Signal naming

Specify rules for naming `Simulink.Signal` and `mpt.Signal` objects in generated code.

Model Configuration Pane: Code Generation / Identifiers

Description

Specify rules for naming `Simulink.Signal` and `mpt.Signal` data objects in generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom MATLAB function` enables **MATLAB function**.
- This parameter must be the same for top-level and referenced models.
- If you give a value to the **Identifier** parameter of a `mpt.Signal` data object, that value overrides the specification of the **Signal naming** parameter.

Settings

None (default) | Force upper case | Force lower case | Custom MATLAB function

None

Does not change signal names when creating corresponding identifiers in generated code. Signal identifiers in the generated code match the signal names that appear in the model.

Force upper case

Uses uppercase characters when creating identifiers for signal names in the generated code.

Force lower case

Uses lowercase characters when creating identifiers for signal names in the generated code.

Custom MATLAB function

Uses the MATLAB function specified with the **MATLAB function** parameter to create identifiers for signal names in the generated code.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Force upper case
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: `SignalNamingRule`

Type: character vector

Value: None | UpperCase | LowerCase | Custom

Default: None

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Apply Custom Naming Conventions to Identifiers”

“Programming”

MATLAB function

Specify rules for naming identifiers in generated code

Model Configuration Pane: Code Generation / Identifiers

Description

Specify rules for naming identifiers in generated code.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **Signal naming**.
- Must be the same for top-level and referenced models.

Settings

' ' (default) | string

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

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

For example, the following function returns an identifier name by appending the text `_signal` to a signal data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_signal';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: SignalNamingFcn

Type: character vector

Value: MATLAB language file

Default: ''

Version History

Introduced before R2006a

R2026a: Renamed from M-function

Behavior changed in R2026a

M-function is now called **MATLAB function** in the model configuration parameters dialog box. The programmatic name SignalNamingFcn remains the same.

See Also

Topics

"Model Configuration Parameters: Code Generation Identifiers" on page 15-2

"Apply Custom Naming Conventions to Identifiers"

"Programming"

Parameter naming

Specify rules for naming `Simulink.Parameter` and `mpt.Parameter` objects in generated code

Model Configuration Pane: Code Generation / Identifiers

Description

Specify rules for naming `Simulink.Parameter` and `mpt.Parameter` data objects in generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to Custom MATLAB function enables **MATLAB function**.
- This parameter must be the same for top-level and referenced models.

Settings

None (default) | Force upper case | Force lower case | Custom MATLAB function

None

Does not change parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case

Uses uppercase characters when creating identifiers for parameter names in the generated code.

Force lower case

Uses lowercase characters when creating identifiers for parameter names in the generated code.

Custom MATLAB function

Uses the MATLAB function specified with the **MATLAB function** parameter to create identifiers for parameter names in the generated code.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Force upper case
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ParamNamingRule

Type: character vector

Value: None | UpperCase | LowerCase | Custom

Default: None

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Apply Custom Naming Conventions to Identifiers”

“Programming”

MATLAB function

Specify rules for naming identifiers in generated code

Model Configuration Pane: Code Generation / Identifiers

Description

Specify rules for naming identifiers in generated code.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **Parameter naming**.
- Must be the same for top-level and referenced models.

Settings

' ' (default) | string

Default: ' '

Enter the name of a MATLAB language file that contains the naming rule to be applied to signal, parameter, or #define parameter identifiers in generated code. Examples of rules you might program in such a MATLAB function include:

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

For example, the following function returns an identifier name by appending the text `_param` to a parameter data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_param';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ParamNamingFcn

Type: character vector

Value: MATLAB language file

Default: ''

Version History

Introduced before R2006a

R2026a: Renamed from M-function

Behavior changed in R2026a

M-function is now called **MATLAB function** in the model configuration parameters dialog box. The programmatic name ParamNamingFcn remains the same.

See Also

Topics

"Model Configuration Parameters: Code Generation Identifiers" on page 15-2

"Apply Custom Naming Conventions to Identifiers"

"Programming"

#define naming

Specify rule for naming `#define` parameters in generated code

Model Configuration Pane: Code Generation / Identifiers

Description

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom MATLAB function` enables **MATLAB function**.
- This parameter must be the same for top-level and referenced models.

Settings

None (default) | Force upper case | Force lower case | Custom MATLAB function

None

Does not change `#define` parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case

Uses uppercase characters when creating identifiers for `#define` parameter names in the generated code.

Force lower case

Uses lowercase characters when creating identifiers for `#define` parameter names in the generated code.

Custom MATLAB function

Uses the MATLAB function specified with the **MATLAB function** parameter to create identifiers for `#define` parameter names in the generated code.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Force upper case
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: DefineNamingRule

Type: character vector

Value: None | UpperCase | LowerCase | Custom

Default: None

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers” on page 15-2

“Apply Custom Naming Conventions to Identifiers”

“Programming”

MATLAB function

Specify rule for naming identifiers in generated code

Model Configuration Pane: Code Generation / Identifiers

Description

Specify rule for naming identifiers in generated code.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **#define naming**.
- Must be the same for top-level and referenced models.

Settings

' ' (default) | string

Enter the name of a MATLAB language file that contains the naming rule to be applied to signal, parameter, or **#define** parameter identifiers in generated code. Examples of rules you might program in such a MATLAB function include:

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

For example, the following function returns an identifier name by appending the text `_define` to a data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a #define data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_define';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: DefineNamingFcn

Type: character vector

Value: MATLAB language file

Default: ''

Version History

Introduced before R2006a

R2026a: Renamed from M-function

Behavior changed in R2026a

M-function is now called **MATLAB function** in the model configuration parameters dialog box. The programmatic name DefineNamingFcn remains the same.

See Also

Topics

"Model Configuration Parameters: Code Generation Identifiers" on page 15-2

"Apply Custom Naming Conventions to Identifiers"

"Programming"

Custom token text

Specify text to insert for \$U token

Model Configuration Pane: Code Generation / Identifiers

Description

Specify text to insert for \$U token.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code

Settings

' ' (default) | string

Enter text to insert for the \$U token.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Set a custom string and use \$U in symbols
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: CustomUserTokenString

Type: character vector

Value: ' ' or user-specified name

Default: ' '

Version History

Introduced in R2018a

See Also

Topics

“Identifier Format Control”

“Model Configuration Parameters: Code Generation Identifiers”

Shared checksum length

Specify character length of \$C token

Model Configuration Pane: Code Generation / Identifiers

Description

Specify character length of \$C token.

Category: Code Generation > Identifiers

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Settings

8 (default) | integer

Minimum: 1

Maximum: 15

Specify an integer value that indicates the number of characters to expand the \$C token for the **Shared utilities identifier format** parameter.

Tip

To avoid the possibility of a naming collision, consider increasing this parameter value.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: SharedChecksumLength

Type: integer

Value: valid value

Default: 8

Version History

Introduced in R2017b

See Also

Topics

"Model Configuration Parameters: Code Generation Identifiers"
Shared utilities identifier format

EMX array utility functions identifier format

Customize generated identifiers for `emxArray` utility functions

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated identifiers for `emxArray` (embeddable `mxArray`) utility functions. The code generator produces `emxArray` types for variable-size arrays that use dynamically allocated memory. It produces `emxArray` utility functions that create and interact with variables that have an `emxArray` type. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object™ associated with a MATLAB System block. This parameter does not apply to:

- Input or output signals
- Parameters
- Global variables
- Discrete state properties of System objects associated with a MATLAB System block

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires an Embedded Coder when generating code.

Settings

`emx$M$N` | string

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 the following format tokens.

Token	Description
\$M	Insert name-mangling text if required to avoid naming collisions. Required.
\$N	Insert the utility function name into identifier. For example, <code>Init_real</code> .
\$R	Insert root model name into identifier, replacing unsupported characters with the underscore (<code>_</code>) character. Required for model referencing.

Tips

- The code generator applies the identifier format specified by this parameter before it applies the formats specified by other identifier format control parameters.

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- If you specify \$R, the value that you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrEmxFcn

Type: character vector

Value: valid combination of tokens

Default: emx\$M\$N

Version History

Introduced in R2017a

See Also

Topics

“Model Configuration Parameters: Code Generation Identifiers”

“Identifier Format Control”

“Avoid Identifier Naming Collisions”

EMX array types identifier format

Customize generated identifiers for `emxArray` types

Model Configuration Pane: Code Generation / Identifiers

Description

Customize generated identifiers for `emxArray` (embeddable `mxArray`) types. The code generator produces `emxArray` types for variable-size arrays that use dynamically allocated memory. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block. This parameter does not apply to:

- Input or output signals
- Parameters
- Global variables
- Discrete state properties of System objects associated with a MATLAB System block

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires an Embedded Coder when generating code.

Settings

`emxArray_``MN` | string

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 the following format tokens.

Token	Description
<code>\$M</code>	Insert name-mangling text if required to avoid naming collisions. Required.
<code>\$N</code>	Insert type name. For example, <code>real_T</code>
<code>\$R</code>	Insert root model name into identifier, replacing unsupported characters with the underscore (<code>_</code>) character. Required for model referencing.

Tips

- The code generator applies the identifier format specified by this parameter before it applies the formats specified by other identifier format control parameters.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.

- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: CustomSymbolStrEmxType

Type: character vector

Value: valid combination of tokens

Default: emxArray_ \$M\$N

Version History

Introduced in R2017a

See Also

Topics

"Model Configuration Parameters: Code Generation Identifiers"

"Identifier Format Control"

"Avoid Identifier Naming Collisions"

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
Data type replacement	Method for replacing Simulink data types in generated code.
Specify custom data type names	Option to replace data type names determined by code generation settings with user-defined data type names in generated code.
Replacement Name: double	Replacement name for <code>double</code> built-in data type in generated code.
Replacement Name: single	Replacement name for <code>single</code> built-in data type in generated code.
Replacement Name: int32	Replacement name for <code>int32</code> built-in data type in generated code.
Replacement Name: int16	Replacement name for <code>int16</code> built-in data type in generated code.
Replacement Name: int8	Replacement name for <code>int8</code> built-in data type in generated code.
Replacement Name: uint32	Replacement name for <code>uint32</code> built-in data type in generated code.
Replacement Name: uint16	Replacement name for <code>uint16</code> built-in data type in generated code.
Replacement Name: uint8	Replacement name for <code>uint8</code> built-in data type in generated code.
Replacement Name: boolean	Replacement name for <code>boolean</code> built-in data type in generated code.
Replacement Name: int	Replacement name for <code>int</code> built-in data type in generated code.
Replacement Name: uint	Replacement name for <code>uint</code> built-in data type in generated code.
Replacement Name: char	Replacement name for <code>char</code> built-in data type in generated code.
Replacement Name: uint64	Replacement name for <code>uint64</code> built-in data type in generated code.
Replacement Name: int64	Replacement name for <code>int64</code> built-in data type in generated code.

These parameters belong to the **Advanced parameters** category.

Parameter	Description
Boolean false identifier	Identifier for Boolean <code>false</code>
Boolean true identifier	Identifier for Boolean <code>true</code> value
64-bit integer maximum identifier	Identifier for 64-bit integer maximum
16-bit integer maximum identifier	Identifier for 16-bit integer maximum
32-bit integer maximum identifier	Identifier for 32-bit integer maximum
8-bit integer maximum identifier	Identifier for 8-bit integer maximum
64-bit unsigned integer maximum identifier	Identifier for 64-bit unsigned integer maximum
16-bit unsigned integer maximum identifier	Identifier for 16-bit unsigned integer maximum
32-bit unsigned integer maximum identifier	Identifier for 32-bit unsigned integer maximum
8-bit unsigned integer maximum identifier	Identifier for 8-bit unsigned integer maximum
64-bit integer minimum identifier	Identifier for 64-bit integer minimum
16-bit integer minimum identifier	Identifier for 16-bit integer minimum
32-bit integer minimum identifier	Identifier for 32-bit integer minimum
8-bit integer minimum identifier	Identifier for 8-bit integer minimum
Type limit identifier replacement header file	Header file that defines Boolean and data type limit identifiers
Implement images using OpenCV Mat class	Option to represent images as OpenCV class <code>cv::Mat</code>
Coder typedefs compatibility	Option to force generation of <code>rtwtypes.h</code> .

Configure Simulink Coder Data Type Replacements Programmatically

To programmatically replace the Simulink Coder 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_B00L`.

```
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_B00L';

set_param(cs, 'ReplacementTypes', struc);
```

See Also

More About

- “Manage Replacement of Simulink Data Types in Generated Code”
- “Specify Boolean and Data Type Limit Identifiers”

- “Model Configuration Set Customization”

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.

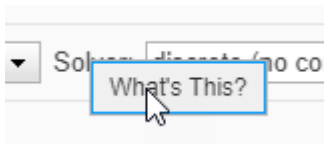
Configuration

This tab is visible only if you specify an ERT-based System target file.

- 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 16-2
- “Manage Replacement of Simulink Data Types in Generated Code”

Data type replacement

Method for replacing built-in data type names in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Data type replacement** parameter specifies the method for replacing Simulink data types in generated code.

Settings

Use C data types with fixed-width integers | Use coder typedefs

Use coder typedefs

The code generator creates the header file `rtwtypes.h`, which specifies Simulink Coder data types that are based on the C89 language standard. The generated code uses Simulink Coder data types for scalar and array variables. For example, the Simulink Coder type `real32_T` corresponds to the Simulink type `single`.

This option is selected if you open:

- A GRT-based model and change **System target file** from `grt.tlc` to `ert.tlc`.
- A model that was last saved by using R2022b or an earlier release.

Use C data types with fixed-width integers

The generated code uses data types from the C99 language standard, which includes definitions from `stdint.h` and `stdbool.h`.

Selecting this option enables the **Coder typedefs compatibility** (CoderTypedefsCompatibility) check box. By default, the code generator does not create `rtwtypes.h` because the generated code does not require definitions from `rtwtypes.h`. The file `rtwtypes.h` is generated in some cases because it might be needed by static source code located under `matlabroot`. If you use custom code that requires Simulink Coder data type definitions, you can force the generation of `rtwtypes.h` by selecting the **Coder typedefs compatibility** check box.

This option is selected if:

- You open an Embedded Coder template model.
- For the **Shared coder dictionary** (EmbeddedCoderDictionary) configuration parameter, you specify a file that contains a service interface configuration.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	Use C data types with fixed-width integers improves compliance with MISRA C and MISRA C++ standards.

Programmatic Use

Parameter: DataTypeReplacement

Type: character vector

Value: 'CoderTypedefs' | 'CDataTypesFixedWidth'

Version History

Introduced in R2023a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

“Specify Boolean and Data Type Limit Identifiers”

Simulink.AliasType

Simulink.NumericType

“Compatibility of TLC Files with Generated Code Data Types”

Specify custom data type names

Option to replace built-in data type names in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Specify custom data type names** parameter specifies whether to replace built-in data type names in the generated code with data type names that you specify.

When you want to generate code that adheres to application or site standards for data type naming, this parameter enables you to specify replacements in the generated code for:

- Built-in data type names through the **Data type names** table.
- Boolean and data type limit identifiers through the **Advanced parameters** section.

Settings

off (default) | on

on

Display the **Data type names** table, which enables you to change the names of built-in data types used by the generated code.

You can specify new data type names for some or all of the 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.

If you use a Simulink.AliasType or Simulink.NumericType object and **Data type replacement** is set to Use C data types with fixed-width integers, the HeaderFile object property must be set to the name of the header file that provides the type definition.

An error occurs, if:

- A replacement data type specification is not consistent with the **Simulink Name** data type.
- The Simulink.AliasType object has the **Data scope** parameter set to Exported.

off

In generated code, use Simulink Coder names for built-in Simulink data types.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: EnableUserReplacementTypes

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

“Specify Boolean and Data Type Limit Identifiers”

Simulink.AliasType

Simulink.NumericType

Coder typedefs compatibility

Option to force generation of `rtwtypes.h` file

Model Configuration Pane: Code Generation / Data Type Replacement

Description

If you set **Data type replacement** to Use C data types with fixed-width integers, the generated code uses data types from the C99 language standard and does not require `rtwtypes.h` definitions. By default, the code generator does not create the `rtwtypes.h` file. The `rtwtypes.h` file is generated in some cases because it might be needed by static source code located under `matlabroot`. If you use custom code that requires Simulink Coder data type definitions, select the **Coder typedefs compatibility** check box to force the generation of the `rtwtypes.h` file.

Dependencies

To enable this parameter, set **Data type replacement** to Use C data types with fixed-width integers.

Settings

off (default) | on

on

Generate the `rtwtypes.h` file.

off

Do not generate the `rtwtypes.h` file.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: `CoderTypedefsCompatibility`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2023a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

Replacement Name: double

Name for double data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: double** parameter specifies the name to use for the built-in data type double in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- `real_T` if **Data type replacement** is Use coder typedefs.
- `double` if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use a `Simulink.AliasType` object:

- Set the `BaseType` object property to `double`.
- Set the `HeaderFile` object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use a `Simulink.NumericType` object:

- Set the `DataTypeMode` object property to `Double`.
- Set the `HeaderFile` object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify `double`.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Manage Replacement of Simulink Data Types in Generated Code"

`Simulink.AliasType`

`Simulink.NumericType`

Replacement Name: single

Name for single data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: single** parameter specifies the name to use for the built-in data type `single` in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- `real32_T` if **Data type replacement** is Use coder typedefs.
- `float` if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use a `Simulink.AliasType` object:

- Set the `BaseType` object property to `single`.
- Set the `HeaderFile` object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use a `Simulink.NumericType` object:

- Set the `DataTypeMode` object property to `Single`.
- Set the `HeaderFile` object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify `single`.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Manage Replacement of Simulink Data Types in Generated Code"

`Simulink.AliasType`

`Simulink.NumericType`

Replacement Name: int32

Name for int32 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: int32** parameter specifies the name to use for the built-in data type int32 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- int32_T if **Data type replacement** is Use coder typedefs.
- int32_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.AliasType object:

- Set the BaseType object property to int32.
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify int32.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

Simulink.AliasType

Replacement Name: int16

Name for int16 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: int16** parameter specifies the name to use for the built-in data type int16 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- int16_T if **Data type replacement** is Use coder typedefs.
- int16_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.AliasType object:

- Set the BaseType object property to int16.
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify int16.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

Simulink.AliasType

Replacement Name: int8

Name for int8 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: int8** parameter specifies the name to use for the built-in data type int8 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- int8_T if **Data type replacement** is Use coder typedefs.
- int8_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.AliasType object:

- Set the BaseType object property to int8.
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify int8.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

Simulink.AliasType

Replacement Name: uint32

Name for uint32 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: uint32** parameter specifies the name to use for the built-in data type uint32 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- uint32_T if **Data type replacement** is Use coder typedefs.
- uint32_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.AliasType object:

- Set the BaseType object property to uint32.
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify uint32.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

Simulink.AliasType

Replacement Name: uint16

Name for uint16 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: uint16** parameter specifies the name to use for the built-in data type uint16 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- uint16_T if **Data type replacement** is Use coder typedefs.
- uint16_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.AliasType object:

- Set the BaseType object property to uint16.
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify uint16.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

Simulink.AliasType

Replacement Name: uint8

Name for uint8 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: uint8** parameter specifies the name to use for the built-in data type uint8 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- uint8_T if **Data type replacement** is Use coder typedefs.
- uint8_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.AliasType object:

- Set the BaseType object property to uint8.
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify uint8.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

“Model Configuration Parameters: Code Generation Data Type Replacement” on page 16-2

“Manage Replacement of Simulink Data Types in Generated Code”

Simulink.AliasType

Replacement Name: boolean

Name for boolean data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: boolean** parameter specifies the name to use for the built-in data type boolean in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- `boolean_T` if **Data type replacement** is Use coder typedefs.
- `bool` if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use a `Simulink.AliasType` object:

- Set the `BaseType` object property to `boolean`, `uint8`, `int8`, or `intn`, where n is the number of bits set for **Configuration Parameters > Hardware Implementation > Number of bits: int**.
- Set the `HeaderFile` object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use a `Simulink.NumericType` object:

- Set the `DataTypeMode` object property to `Boolean`.
- Set the `HeaderFile` object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the name of the `Simulink.NumericType` object.

To use the **Simulink Name**:

- In the **Replacement Name** column, specify `uint8`, `int8`, or `intn`, where n is the number of bits set for **Configuration Parameters > Hardware Implementation > Number of bits: int**.

For ERT S-functions, the replacement data type can be only an 8-bit integer, `int8`, or `uint8`.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Replace boolean with Specific Integer Data Type"

"Manage Replacement of Simulink Data Types in Generated Code"

Simulink.AliasType

Simulink.NumericType

Replacement Name: int

Name for int data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: int** parameter specifies the name to use for the built-in data type int in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- `int_T` if **Data type replacement** is Use coder typedefs.
- `int` if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To use a `Simulink.AliasType` object:

- Set the `BaseType` object property to `intn`.
- Set the `HeaderFile` object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the `intn`.

To use the **Simulink Name**, in the **Replacement Name** column, specify `intn`.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Manage Replacement of Simulink Data Types in Generated Code"

Simulink.AliasType

Replacement Name: uint

Name for uint data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: uint** parameter specifies the name to use for the built-in data type uint in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- uint_T if **Data type replacement** is Use coder typedefs.
- unsigned int32 if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.AliasType object:

- Set the BaseType object property to uint n .
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify uint n .

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

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: ''

Version History

Introduced in R2007a

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Manage Replacement of Simulink Data Types in Generated Code"

Simulink.AliasType

Replacement Name: char

Name for char data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: char** parameter specifies the name to use for the built-in data type char in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- char_T if **Data type replacement** is Use coder typedefs.
- char if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using a Simulink.AliasType object:

- Set the BaseType object property to int*n*. *n* is the number of bits set for **Configuration Parameters > Hardware Implementation > Number of bits: char**.
- Set the HeaderFile object property to the name of the header file that provides the type definition.
- In the **Replacement Name** column, specify the name of the object.

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: ReplacementTypes, replacementName.char

Type: character vector

Value: The name of a Simulink.AliasType object, where the object exists in the base workspace.

Default: ''

Version History

Introduced in R2007a

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Manage Replacement of Simulink Data Types in Generated Code"

Simulink.AliasType

Replacement Name: uint64

Name for uint64 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: uint64** parameter specifies the name to use for the built-in data type uint64 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- uint64_T if **Data type replacement** is Use coder typedefs.
- uint64_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.NumericType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.NumericType object:

- Set these properties:
 - DataTypeMode -- Fixed-point: binary point scaling
 - Signedness -- Unsigned
 - WordLength -- 64
 - IsAlias -- true
 - HeaderFile -- Name of header file that provides type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify uint64.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The object is Simulink.AliasType.

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	A valid character vector
Efficiency	Noimpact
Safety precaution	No recommendation

Programmatic Use

Parameter: ReplacementTypes, replacementName.uint64

Type: character vector

Value: The name of a Simulink.NumericType object, where the object exists in the base workspace.

Default: ''

Version History

Introduced in R2018b

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Manage Replacement of Simulink Data Types in Generated Code"

Simulink.NumericType

Replacement Name: int64

Name for int64 data type in generated code

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Replacement Name: int64** parameter specifies the name to use for the built-in data type int64 in the generated code.

Dependencies

To enable this parameter, select the Specify custom data type names parameter.

Settings

no default

If a value is not specified, the code generator uses the **Code Generation Name:**

- int64_T if **Data type replacement** is Use coder typedefs.
- int64_t if **Data type replacement** is Use C data types with fixed-width integers.

You can specify a replacement name by using one of the following:

- A Simulink.NumericType object.
- The **Simulink Name** built-in data type name.

To use a Simulink.NumericType object:

- Set these properties:
 - DataTypeMode -- Fixed-point: binary point scaling
 - Signedness -- Signed
 - WordLength -- 64
 - IsAlias -- true
 - HeaderFile -- Name of header file that provides type definition.
- In the **Replacement Name** column, specify the object name.

To use the **Simulink Name**, in the **Replacement Name** column, specify int64.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The object is Simulink.AliasType.

Recommended Settings

Application	Setting
Debugging	Noimpact
Traceability	A valid character vector
Efficiency	Noimpact
Safety precaution	No recommendation

Programmatic Use

Parameter: ReplacementTypes, replacementName.int64

Type: character vector

Value: The name of a Simulink.NumericType object, where the object exists in the base workspace.

Default: ''

Version History

Introduced in R2018b

See Also

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

"Manage Replacement of Simulink Data Types in Generated Code"

Simulink.NumericType

Boolean false identifier

Identifier for Boolean `false`

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default Boolean `false` identifier in the generated code.

Settings

`false` (default) | character vector

Use a valid MATLAB variable name as a replacement identifier.

Examples

Replace Boolean `false` Identifier

For an example that shows how to specify a replacement identifier for Boolean `false`, see “Boolean Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: BooleanFalseId

Type: character vector

Values: 'false' or valid MATLAB variable name

Default: 'false'

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

Boolean true identifier

Identifier for Boolean `true`

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default Boolean `true` identifier in the generated code.

Settings

`true` (default) | character vector

Use a valid MATLAB variable name as a replacement identifier.

Examples

Replace Boolean `true` Identifier

For an example that shows how to specify a replacement identifier for Boolean `true`, see “Boolean Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: BooleanTrueId

Type: character vector

Values: 'true' or valid MATLAB variable name

Default: 'true'

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

64-bit integer maximum identifier

Identifier for 64-bit integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 64-bit integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- INT64_MAX if **Data type replacement** is Use C data types with fixed-width integers
- MAX_int64_T if **Data type replacement** is Use coder typedefs

Examples

Replace 64-Bit Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MaxIdInt64

Type: character vector

Values: valid MATLAB variable name

Default: depends on DataTypeReplacement

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

16-bit integer maximum identifier

Identifier for 16-bit integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 16-bit integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- INT16_MAX if **Data type replacement** is Use C data types with fixed-width integers
- MAX_int16_T if **Data type replacement** is Use coder typedefs

Examples

Replace 16-Bit Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MaxIdInt16

Type: character vector

Values: valid MATLAB variable name

Default: depends on DataTypeReplacement

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

32-bit integer maximum identifier

Identifier for 32-bit integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 32-bit integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- INT32_MAX if **Data type replacement** is Use C data types with fixed-width integers
- MAX_int32_T if **Data type replacement** is Use coder typedefs

Examples

Replace 32-Bit Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MaxIdInt32

Type: character vector

Values: valid MATLAB variable name

Default: depends on DataTypeReplacement

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

8-bit integer maximum identifier

Identifier for 8-bit integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 8-bit integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- INT8_MAX if **Data type replacement** is Use C data types with fixed-width integers
- MAX_int8_T if **Data type replacement** is Use coder typedefs

Examples

Replace 8-Bit Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MaxIdInt8

Type: character vector

Values: valid MATLAB variable name

Default: depends on DataTypeReplacement

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

64-bit unsigned integer maximum identifier

Identifier for 64-bit unsigned integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 64-bit unsigned integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- `UINT64_MAX` if **Data type replacement** is Use C data types with fixed-width integers.
- `MAX_uint64_T` if **Data type replacement** is Use coder typedefs.

Examples

Replace 64-Bit Unsigned Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: `MaxIdUInt64`

Type: character vector

Values: valid MATLAB variable name

Default: depends on `DataTypeReplacement`

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

16-bit unsigned integer maximum identifier

Identifier for 16-bit unsigned integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 16-bit unsigned integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- `UINT16_MAX` if **Data type replacement** is Use C data types with fixed-width integers.
- `MAX_uint16_T` if **Data type replacement** is Use coder typedefs.

Examples

Replace 16-Bit Unsigned Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: `MaxIdUint16`

Type: character vector

Values: valid MATLAB variable name

Default: depends on `DataTypeReplacement`

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

32-bit unsigned integer maximum identifier

Identifier for 32-bit unsigned integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 32-bit unsigned integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- `UINT32_MAX` if **Data type replacement** is Use C data types with fixed-width integers.
- `MAX_uint32_T` if **Data type replacement** is Use coder typedefs.

Examples

Replace 32-Bit Unsigned Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: `MaxIdUInt32`

Type: character vector

Values: valid MATLAB variable name

Default: depends on `DataTypeReplacement`

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

8-bit unsigned integer maximum identifier

Identifier for 8-bit unsigned integer maximum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 8-bit unsigned integer maximum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- `UINT8_MAX` if **Data type replacement** is Use C data types with fixed-width integers.
- `MAX_uint8_T` if **Data type replacement** is Use coder typedefs.

Examples

Replace 8-Bit Unsigned Integer Maximum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: `MaxIdUInt8`

Type: character vector

Values: valid MATLAB variable name

Default: depends on `DataTypeReplacement`

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

64-bit integer minimum identifier

Identifier for 64-bit integer minimum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 64-bit integer minimum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- INT64_MIN if **Data type replacement** is Use C data types with fixed-width integers
- MIN_int64_T if **Data type replacement** is Use coder typedefs

Examples

Replace 64-Bit Integer Minimum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MinIdInt64

Type: character vector

Values: valid MATLAB variable name

Default: depends on DataTypeReplacement

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

16-bit integer minimum identifier

Identifier for 16-bit integer minimum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 16-bit integer minimum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- INT16_MIN if **Data type replacement** is Use C data types with fixed-width integers
- MIN_int16_T if **Data type replacement** is Use coder typedefs

Examples

Replace 16-Bit Integer Minimum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MinIdInt16

Type: character vector

Values: valid MATLAB variable name

Default: depends on DataTypeReplacement

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

32-bit integer minimum identifier

Identifier for 32-bit integer minimum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 32-bit integer minimum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- INT32_MIN if **Data type replacement** is Use C data types with fixed-width integers
- MIN_int32_T if **Data type replacement** is Use coder typedefs

Examples

Replace 32-Bit Integer Minimum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: MinIdInt32

Type: character vector

Values: valid MATLAB variable name

Default: depends on DataTypeReplacement

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

8-bit integer minimum identifier

Identifier for 8-bit integer minimum

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The parameter specifies a replacement identifier for the default 8-bit integer minimum identifier in the generated code.

Settings

character vector

Use a valid MATLAB variable name as a replacement identifier.

If you do not specify a value, the code generator uses:

- `INT8_MIN` if **Data type replacement** is Use C data types with fixed-width integers
- `MIN_int8_T` if **Data type replacement** is Use coder typedefs

Examples

Replace 8-Bit Integer Minimum Identifier

For an example that shows how to specify a replacement identifier, see “Data Type Limit Identifiers”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: `MinIdInt8`

Type: character vector

Values: valid MATLAB variable name

Default: depends on `DataTypeReplacement`

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

Type limit identifier replacement header file

Header file that defines Boolean and data type limit identifiers

Model Configuration Pane: Code Generation / Data Type Replacement

Description

Use the parameter to import a header file that defines replacement Boolean and data type limit identifiers for generated code.

Settings

empty (default) | character vector

If you specify a header file that defines replacement Boolean and data type limit identifiers, the code generator creates an `rtwtypes.h` file that includes the header file.

Examples

Import Identifiers Header File

For an example that shows how to import a header file defining Boolean and data type limit identifiers, see “Boolean and Data Type Limit Identifier Header Files”.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid character vector
Efficiency	No impact
Safety precaution	No recommendation

Programmatic Use

Parameter: `TypeLimitIdReplacementHeaderFile`

Type: character vector

Values: valid MATLAB variable name

Default: empty

Version History

Introduced in R2007a

See Also

Topics

“Specify Boolean and Data Type Limit Identifiers”

Implement images using OpenCV Mat class

Option to represent images as OpenCV Mat class

Model Configuration Pane: Code Generation / Data Type Replacement

Description

The **Implement images using OpenCV Mat class** parameter specifies whether to represent images as OpenCV class `cv::Mat`.

Dependencies

This parameter is used to generate C++ code only.

Settings

off (default) | on

on

Generates C++ code where images are represented by using OpenCV class `cv::MAT`.

off

Generates C++ code where images are represented by using C++ class `images::datatypes::Image` implemented by The MathWorks®.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: `ImplementImageWithCVMat`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2021b

See Also

`Simulink.ImageType`

Topics

"Model Configuration Parameters: Code Generation Data Type Replacement" on page 16-2

Memory Sections Parameters on the Code Generation Pane

Code Generation: Memory Sections Tab Overview

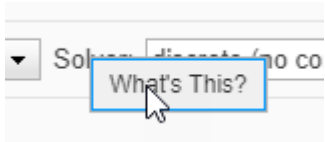
Insert comments and pragmas into the generated code for data and functions.

Configuration

This tab appears only if you specify an ERT based System target file.

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"
- "Control Data and Function Placement in Memory by Inserting Pragmas"

Package

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary. This parameter might be applicable for applying software address methods to AUTOSAR internal data in models configured for the AUTOSAR Classic Platform.

Description

Specify a package that contains memory sections you want to apply to functions and data.

Category: Code Generation

Settings

Memory section specifications for functions and 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

Note The command-line setting '--- None ---' includes spaces.

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”
- “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”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Initialize/Terminate

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary.

Description

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

Category: Code Generation

Settings

Memory section specifications for functions and 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”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Execution

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary.

Description

Specify whether to apply a memory section to execution functions.

Category: Code Generation

Settings

Memory section specifications for functions and 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
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”

- “Control Data and Function Placement in Memory by Inserting Pragmas”

Shared utility

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary.

Description

Specify whether to apply memory sections to shared utility functions.

Category: Code Generation

Settings

Memory section specifications for functions and 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, binary search functions, and MATLAB functions defined outside MATLAB Function blocks.

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”

- “Control Data and Function Placement in Memory by Inserting Pragmas”

Constants

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary.

Description

Specify whether to apply a memory section to constants.

Category: Code Generation

Settings

Memory section specifications for functions and 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 “Data Structures in the Generated Code”.

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”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Inputs/Outputs

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary.

Description

Specify whether to apply a memory section to root input and output.

Category: Code Generation

Settings

Memory section specifications for functions and 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 “Data Structures in the Generated Code”.

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”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Internal data

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary. This parameter might be applicable for applying software address methods to AUTOSAR internal data in models configured for the AUTOSAR Classic Platform.

Description

Specify whether to apply a memory section to internal data.

Category: Code Generation

Settings

Memory section specifications for functions and 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”.

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

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation”
- “Control Data and Function Placement in Memory by Inserting Pragmas”

Parameters

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary.

Description

Specify whether to apply a memory section to parameters.

Category: Code Generation

Settings

Memory section specifications for functions and 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”.

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”
- Memory Sections

Validation results

Note This parameter is not applicable for C code generation. To replicate this functionality for C code generation, use the Code Mappings Editor — C and Embedded Coder Dictionary.

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”

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	Specify the code generation template (CGT) file to use when generating a source code file.
Code templates: Header file (*.h) template	Specify the code generation template (CGT) file to use when generating a code header file.
Data templates: Source file (*.c) template	Specify the code generation template (CGT) file to use when generating a data source file.
Data templates: Header file (*.h) template	Specify the code generation template (CGT) file to use when generating a data header file.
File customization template	Specify the custom file processing (CFP) template file to use when generating code.
Generate an example main program	Control whether to generate an example main program for a model.
Target operating system	Specify a target operating system to use when generating model-specific example main program module.

The following parameters in the **Advanced parameters** section are infrequently used.

Parameter	Description
Generate full file banner	Generate full header including time stamp. For GRT targets, this parameter is on the Code Generation > Interface pane.
Enable custom file banner	If this option is cleared, the configurations for Code and Data templates are ignored.

See Also

More About

- “Model Configuration Set Customization”

Code Generation: Templates Tab Overview

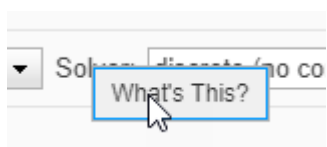
Customize the organization of your generated code.

Configuration

This tab appears only if you specify an ERT based System target file.

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 18-2

Code templates: Source file (*.c) template

Source code file template

Model Configuration Pane: Code Generation / Templates

Description

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

Settings

ert_code_template.cgt (default) | CGT file

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ERTSrcFileBannerTemplate

Type: character vector

Value: valid CGT file

Default: 'ert_code_template.cgt'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2

Selecting and Defining Templates

Custom File Processing

Code templates: Header file (*.h) template

Template for generated code header

Model Configuration Pane: Code Generation / Templates

Description

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

Settings

ert_code_template.cgt (default) | CGT file

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ERTHdrFileBannerTemplate

Type: character vector

Value: valid CGT file

Default: 'ert_code_template.cgt'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2

Selecting and Defining Templates

Custom File Processing

Data templates: Source file (*.c) template

Template for generated data source file

Model Configuration Pane: Code Generation / Templates

Description

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

Settings

ert_code_template.cgt (default) | CGT file

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ERTDataSrcFileTemplate

Type: character vector

Value: valid CGT file

Default: 'ert_code_template.cgt'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2

Selecting and Defining Templates

Custom File Processing

Data templates: Header file (*.h) template

Template for generated data header file

Model Configuration Pane: Code Generation / Templates

Description

Specify the code generation template (CGT) file to use when generating a data header file.

Settings

ert_code_template.cgt (default) | CGT file

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ERTDataHdrFileTemplate

Type: character vector

Value: valid CGT file

Default: 'ert_code_template.cgt'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2

Selecting and Defining Templates

Custom File Processing

File customization template

Template for custom file processing

Model Configuration Pane: Code Generation / Templates

Description

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

Settings

'example_file_process.tlc' (default) | TLC file

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ERTCustomFileTemplate

Type: character vector

Value: valid TLC file

Default: 'example_file_process.tlc'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2
Selecting and Defining Templates
Custom File Processing

Generate an example main program

Control generation of example main program

Model Configuration Pane: Code Generation / Templates

Description

Control whether to generate an example main program for model. This parameter is not relevant and is disabled for component models configured with a service interface.

Dependencies

This parameter enables **Target operating system**.

Settings

on (default) | off

Default: on

On

Generates an example main program, `ert_main.c` (or `.cpp`) for a model. 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 the model:

- Is single-rate or multirate
- Solver mode is set to `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 the static main file as a template for developing embedded applications.

Tips

- After you generate and customize the main program, clear this parameter 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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: GenerateSampleERTMain

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2

“Deploy Applications to Target Hardware”

Custom File Processing

Target operating system

Target operating system for example main program

Model Configuration Pane: Code Generation / Templates

Description

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

Dependencies

- This parameter is enabled by **Generate an example main program**.
- This parameter must be the same for top-level and referenced models.

Settings

BareBoardExample (default) | NativeThreadsExample

Default: BareBoardExample

BareBoardExample

Generates a bareboard main program designed to run under control of a real-time clock, without a 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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: TargetOS

Type: character vector

Value: 'BareBoardExample' | 'NativeThreadsExample'

Default: 'BareBoardExample'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2

“Deploy Applications to Target Hardware”

Custom File Processing

Generate full file banner

Include code generation information in header file banner

Model Configuration Pane: Code Generation / Templates

Description

Specify whether to include additional code generation information in the header file banner.

Settings

on (default) | off

Default: on

On

Includes additional code generation information in the banner of the generated header file. The additional information includes:

- Date and timestamp of code generation
- Selected target for code generation
- Selected hardware for code generation
- Code generation objectives
- Validation result

The banner also includes this basic information:

- Name of the header file
- Name of the model for which you generated code
- Model version number
- Simulink Coder version, release number, and release date

Off

Does not include code generation details in the generated header file banner. The banner still includes:

- Name of the header file
- Name of the model for which you generated code
- Model version number
- Simulink Coder version, release number, and release date

Examples

Control Information in Header File Banner

Compare the banner in the generated header file when you set **Generate full file banner** to different values. For this example, use the model RollAxisAutopilot.

Here is the header file banner that generated using the default **Generate full file banner** value on:

```
/*
 * RollAxisAutopilot.h
 *
 * Code generation for model "RollAxisAutopilot".
 *
 * Model version          : 6.0
 * Simulink Coder version : 9.8 (R2022b) 13-May-2022
 * C source code generated on : Mon Jun 13 13:31:11 2022
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *   1. Execution efficiency
 *   2. Traceability
 *   3. Safety precaution
 *   4. RAM efficiency
 * Validation result: Not run
 */
```

Here is the header file banner that generated using the **Generate full file banner** value off:

```
/*
 * RollAxisAutopilot.h
 *
 * Code generation for model "RollAxisAutopilot".
 *
 * Model version          : 6.0
 * Simulink Coder version : 9.8 (R2022b) 13-May-2022
 *
 */
```

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: GenerateFullHeader

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2
Custom File Processing

Enable custom file banner

Control whether to use configurations for code and data templates

Model Configuration Pane: Code Generation / Templates

Description

Specify whether the code generator uses the code templates specified by the **Code templates** and **Data templates** parameters.

Settings

on (default) | off

Default: on

On

Generate file banners using the code templates specified by the **Code templates** and **Data templates** parameters.

Off

Generate default file banners. When you select **off**, the code generator does not use the code templates specified by the **Code templates** and **Data templates** parameters.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: ERTCustomFileBanners

Type: character vector

Value: 'on' | 'off'

Default: 'on'

Version History

Introduced before R2006a

See Also

Topics

“Model Configuration Parameters: Code Generation Templates” on page 18-2

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	Measure execution times and generate metrics for tasks in generated code.
Measure function execution times	Measure execution times and generate metrics for functions inside generated code.
Workspace variable	Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.
Save options	Specify whether to save code profiling measurement and analysis data to base workspace.
Measure task stack usage	Measure stack usage for tasks in generated code.
Stack workspace variable	Specify workspace variable that collects stack usage measurements and allows viewing and analysis of stack usage profiles.
Third-party tool	Specify a third-party tool for code coverage.
Enable portable word sizes	Allow portability across host and target processors that support different word sizes.
Enable source-level debugging for SIL or PIL	Allow debugging of generated code during a SIL simulation.

This parameter belongs to the **Advanced parameters** category.

Parameter	Description
Create block	Generate a SIL or PIL block.

See Also

More About

- “Software-in-the-Loop Simulation”
- “Processor-in-the-Loop Simulation”
- “Code Execution-Time Profiling”
- “Code Coverage”
- “Model Configuration Set Customization”

Code Generation: Verification Tab Overview

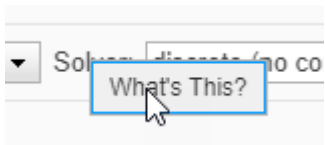
Create SIL block and configure word size portability, code coverage for SIL testing, and code execution profiling

Configuration

This tab appears only if you specify an ERT-based System target file.

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 19-2
- “SIL and PIL Simulations”

Measure task execution time

Execution-time profiling of generated code

Model Configuration Pane: Code Generation / Verification

Description

Measure execution times and generate metrics for tasks in generated code.

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.

Settings

off (default) | on

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 execution-time measurements.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

Programmatic Use

Parameter: CodeExecutionProfiling

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2011a

See Also

Topics

“Model Configuration Parameters: Code Generation Verification” on page 19-2

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

Measure function execution times

Execution-time profiling of functions in generated code

Model Configuration Pane: Code Generation / Verification

Description

Measure execution times and generate metrics for functions inside 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.

Settings

Off (default) | Coarse (referenced models and subsystems only) | Detailed (all function call sites)

Default:

Off

Execution times for functions in generated code are not measured.

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.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

Programmatic Use

Parameter: CodeProfilingInstrumentation

Type: character vector

Value: 'off' | 'coarse' | 'detailed'

Default: 'off'

Version History

Introduced in R2011b

See Also

Topics

"Model Configuration Parameters: Code Generation Verification" on page 19-2

"Create Execution-Time Profile for Generated Code"

"View and Compare Code Execution Times"

"Analyze Code Execution Data"

Workspace variable

Workspace variable for execution-time measurements

Model Configuration Pane: Code Generation / Verification

Description

Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.

Dependencies

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Settings

executionProfile (default) | valid MATLAB variable name

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.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: CodeExecutionProfileVariable

Type: character vector

Value: valid MATLAB variable name

Default: none

Version History

Introduced in R2011a

See Also

Topics

"Model Configuration Parameters: Code Generation Verification" on page 19-2

"Create Execution-Time Profile for Generated Code"

"View and Compare Code Execution Times"

"Analyze Code Execution Data"

Save options

Options for saving execution-time measurements

Model Configuration Pane: Code Generation / Verification

Description

Specify option for saving code profiling measurement and analysis data.

Dependencies

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Settings

Summary data only (default) | All data | Metrics only

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 the base workspace. For example, if you are concerned that your computer might 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 option enables 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 the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` methods.

Selecting this option enables the streaming of execution times to the Simulation Data Inspector during simulations.

Metrics only

During the simulation, store only these profiling metrics on the target hardware:

- Maximum execution time of code section
- Average execution time of code section
- Number of calls to code section

At the end of the simulation, Simulink uploads the stored metrics from the target hardware to your development computer.

The storage of profiling data on the target hardware during the simulation reduces bandwidth usage for the communication channel between Simulink and the target application.

This option does not support:

- Streaming of execution times to the Simulation Data Inspector.
- Overhead filtering.

Recommended Settings

Application	Setting
Debugging	All data
Traceability	All data
Efficiency	Summary data only
Safety precaution	No impact

Programmatic Use

Parameter: CodeProfilingSaveOptions

Type: character vector

Value: 'SummaryOnly' | 'AllData' | 'MetricsOnly'

Default: 'SummaryOnly'

Version History

Introduced in R2011b

See Also

Topics

“Model Configuration Parameters: Code Generation Verification” on page 19-2

“Create Execution-Time Profile for Generated Code”

“View and Compare Code Execution Times”

“Analyze Code Execution Data”

“Remove Instrumentation Overheads from Execution Time Measurements”

“Manage Bandwidth Usage and Memory Allocation During Code Execution Profiling”

Measure task stack usage

Stack usage profiling of generated code

Model Configuration Pane: Code Generation / Verification

Description

Measure stack usage for tasks in generated code.

Dependencies

When you use this parameter, you must also specify a stack workspace variable. The software uses this variable to collect stack usage 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.

Settings

off (default) | on

Default: off

On

During SIL and PIL simulations, collect stack usage measurements for tasks. The software obtains data from instrumentation in the SIL or PIL application.

Off

Do not collect measurements of stack usage

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No recommendation

Programmatic Use

Parameter: CodeStackProfiling

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2022a

See Also

Topics

“Model Configuration Parameters: Code Generation Verification” on page 19-2

“View and Compare Stack Usage Metrics”

Stack workspace variable

Workspace variable for stack usage measurements

Model Configuration Pane: Code Generation / Verification

Description

Specify workspace variable that collects stack usage measurements and allows viewing and analysis of stack usage profiles.

Dependencies

You can only specify this parameter if you select the **Measure task stack usage** check box. Otherwise the field appears dimmed.

Settings

stackProfile (default) | valid MATLAB variable name

Default: stackProfile

When you run simulation, software generates specified workspace variable, which contains stack usage measurements.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact
Safety precaution	No impact

Programmatic Use

Parameter: CodeStackProfileVariable

Type: character vector

Value: valid MATLAB variable name

Default: 'stackProfile'

Version History

Introduced in R2022a

See Also

Topics

“Model Configuration Parameters: Code Generation Verification” on page 19-2

“View and Compare Stack Usage Metrics”

Third-party tool

Tool for code coverage analysis

Model Configuration Pane: Code Generation / Verification

Description

Specify a third-party tool for code coverage analysis of generated code.

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.

Settings

None (use Simulink Coverage) (default) | LDRACover or LDRA tool suite

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.

LDRACover or LDRA tool suite

Specifies the LDRA tool suite from LDRA Software Technology

Recommended Settings

Application	Setting
Debugging	LDRACover or LDRA tool suite
Traceability	LDRACover or LDRA tool suite
Efficiency	None (code coverage off)
Safety precaution	No recommendation

Programmatic Use

Parameter: CoverageTool field of CodeCoverageSettings

Type: character vector

Value: 'None' || 'LDRACover or LDRA tool suite'

Default: 'None'

Version History

Introduced in R2010a

See Also

Topics

“Model Configuration Parameters: Code Generation Verification” on page 19-2

“Configure Code Coverage with Third-Party Tool”

“Configure Code Coverage Programmatically”

Enable portable word sizes

Portability across development computer and target processors

Model Configuration Pane: Code Generation / Verification

Description

Generate conditional processing macros to support compilation of generated code on your MATLAB development computer, which can have a different word size than the target processor on which production code is run. This option enables you to use the same generated code for SIL testing on your MATLAB development computer and for production deployment on the target processor. For example, you can perform SIL testing on a 64-bit development computer and deploy the code on a 16-bit target processor.

Dependencies

When you use this option, you must select **Test hardware is the same as production hardware** on the **Hardware Implementation** pane.

Settings

off (default) | on

Default: off

On

Generate portable code

Off

Does not generate portable code.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No impact

Programmatic Use

Parameter: PortableWordSizes

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2007a

See Also

Topics

"Model Configuration Parameters: Code Generation Verification" on page 19-2

"Configure Hardware Implementation Settings"

"SIL Simulations with Target-Specific Code"

Enable source-level debugging for SIL or PIL

Generated code debugging during SIL or PIL simulation

Model Configuration Pane: Code Generation / Verification

Description

Run SIL or PIL simulations with a debugger enabled. Use the debugger to observe and understand the behavior of generated code sections.

Settings

off (default) | on

Default: off

On

Enable source-level debugging.

Off

Disable source-level debugging.

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No impact

Programmatic Use

Parameter: SILPILDebugging

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Version History

Introduced in R2013a

See Also

Topics

“Model Configuration Parameters: Code Generation Verification” on page 19-2

“Debug Generated Code During SIL or PIL Simulation”

Create block

SIL or PIL block creation

Model Configuration Pane: Code Generation / Verification

Description

Generate a SIL or PIL block

Settings

None (default) | SIL | PIL

Default: None

None

Do not generate SIL or PIL block.

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 your development computer. With this block, you can test the 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 test the 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.

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

Programmatic Use

Parameter: CreateSILPILBlock

Type: character vector

Value: 'None' | 'SIL' | 'PIL'

Default: 'None'

Version History

Introduced in R2007a

See Also

Topics

“SIL and PIL Simulations”

“Create PIL Target Connectivity Configuration for Simulink”

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”. 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)	No impact	No impact	Optimal finite value	inf	auto
Optimize using the specified minimum and maximum values	Off	Off	On	No impact	Off
Remove root level I/O zero initialization	No impact	No impact	On (GUI) off (command line) (execution, ROM), No impact (RAM)	No recommendation	On
Remove internal data zero initialization	No impact	No impact	On (execution, ROM)	No recommendation	On
Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values	Off	Off	On (execution, ROM)	No recommendation	On
Remove code that protects against integer division arithmetic exceptions	No impact	No impact	On (execution, ROM)	Off	Off
Pack Boolean data into bitfields	No impact	No Impact	Off (execution, ROM), On (RAM)	No impact	Off
Pass reusable subsystem outputs as	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	No impact	No impact	No impact	No impact	- - -None - - -
Initialize/ Terminate	No impact	No impact	No impact	No impact	Default
Execution	No impact	No impact	No impact	No impact	Default
Shared utility	No impact	No impact	No impact	No impact	Default
Constants	No impact	No impact	No impact	No impact	Default
Inputs/Outputs	No impact	No impact	No impact	No impact	Default
Internal data	No impact	No impact	No impact	No impact	Default
Parameters	No impact	No impact	No impact	No impact	Default
Validation results	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	On	On	No impact	No recommendation	Off
Model-to-code	On	On	No impact	No recommendation	Off
Generate model Web view	No impact	No impact	No impact	No impact	Off
Eliminated / virtual blocks	On	On	No impact	No recommendation	Off
Traceable Simulink blocks	On	On	No impact	No recommendation	Off
Traceable Stateflow objects	On	On	No impact	No recommendation	Off
Traceable MATLAB functions	On	On	No impact	No recommendation	Off
Generate static code metrics	No impact	No impact	No impact	No impact	Off
Summarize which blocks triggered code replacements	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	On	On	No impact	No impact	On
Simulink data object descriptions 9932000	On	On	No impact	No impact	On
Custom comments (MPT objects only)	On	On	No impact	No impact	Off
Custom comments function	Valid file name	Valid file name	No impact	No impact	' '
Stateflow object descriptions	On	On	No impact	No impact	On
Requirements in block comments	On	On	No impact	No recommendation	Off

Mapping of Application Requirements to the Code Generation Pane: Identifiers Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Global variables	No impact	Use default	No impact	No recommendation	\$R\$N\$M
Global types	No impact	Use default	No impact	No recommendation	&N\$R\$M_T
Field name of global types	No impact	Use default	No impact	No recommendation	\$N\$M
Subsystem methods	No impact	Use default	No impact	No recommendation	\$R\$N\$M\$F
Subsystem method arguments	No impact	Use default	No impact	No recommendation	rt\$I\$N\$M
Local temporary variables	No impact	Use default	No impact	No recommendation	\$N\$M
Local block output variables	No impact	Use default	No impact	No recommendation	rtb_\$N\$M
Constant macros	No impact	Use default	No impact	No recommendation	\$R\$N\$M
Shared utilities identifier format	No impact	Use default	No impact	No recommendation	\$N\$C
Minimum mangle length	No impact	1	No impact	No impact	1
Maximum identifier length	Valid value	>30	No impact	>30	31
System-generated identifiers	No impact	No impact	No impact	No impact	Shortened
Generate scalar inlined parameters as	No impact	Macros	Literals	No impact	Literals
Use the same reserved names as Simulation Target	No impact	No impact	No impact	No impact	Off
Shared checksum length	No impact	No impact	No impact	No impact	8
EMX array utility functions identifier format	No impact	No impact	No impact	No recommendation	emx\$M\$N
EMX array types identifier format	No impact	No impact	No impact	No recommendation	emxArray_\$M\$N

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Custom token text	No impact	Set a custom string and use \$U in symbols	No impact	No impact	' '
#define naming	No impact	Force uppercase	No impact	No impact	None
Parameter naming	No impact	Force uppercase	No impact	No impact	None
Signal naming	No impact	Force uppercase	No impact	No impact	None
MATLAB function	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	No impact	No impact	Off (GUI), 'on' (command-line) for integer only	No impact	On (GUI), 'off' (command-line)
Support: complex numbers	No impact	No impact	Off for real only	No impact	On
Support: absolute time	No impact	No impact	Off	No recommendation	On
Support: continuous time	No impact	No impact	Off (execution, ROM), No impact (RAM)	No recommendation	Off
Support non-inlined S-functions	No impact	No impact	Off	No recommendation	Off
Support: variable-size signals	No impact	No impact	No impact	No recommendation	Off
Multiword type definitions	No impact	No impact	No impact	No recommendation	System defined
Maximum word length	No impact	No impact	No impact	No recommendation	256 for ERT targets 2048 for GRT targets
Pass root-level I/O as	No impact	No impact	No impact	No impact	Individual arguments
Use dynamic memory allocation for model initialization	No impact	No impact	No impact	No recommendation	Off
Terminate function required	No impact	No impact	No impact	No recommendation	On
Remove error status field in real-time model data structure	Off	No impact	On	No recommendation	Off
Include model types in model class	No impact	No impact	On	No recommendation	On
Combine signal/state structures	Off	No impact	No impact	On	No impact

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Generate destructor	No impact	No impact	No impact	No recommendation	On
Use dynamic memory allocation for model block instantiation	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	On	Off	No recommendation	Off
Measure function execution times	On	On	Off	No recommendation	Off
Workspace variable	No impact	Valid MATLAB variable name	No impact	No impact	Off
Save options	All data	All data	Summary data only	No impact	Summary data only
Measure task stack usage	On	On	Off	No recommendation	Off
Stack workspace variable	No impact	Valid MATLAB variable name	No impact	No impact	Off
Third-party tool	LDRAcover or LDRA tool suite	LDRAcover or LDRA tool suite	None (code coverage off)	No recommendation	None (code coverage off)
Enable portable word sizes	On	On	Off	No impact	Off
Enable source-level debugging for SIL or PIL	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	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	On	Off	No recommendation	Off
Preserve condition expression in if statement	On	On	Off	No recommendation	Off
Convert if-elseif-else patterns to switch-case statements	No impact	Off	On (execution, ROM), No impact (RAM)	No impact	On
Preserve extern keyword in function declarations	No impact	No impact	No impact	No impact	On
Preserve static keyword in function declarations	No impact	No impact	On (execution, ROM)	No impact	On
Suppress generation of default cases for switch statements if unreachable	On	On	Off	No recommendation	On
Replace multiplications by powers of two with signed bitwise shifts	No impact	No impact	On	No impact	On
Casting modes	Nominal	Nominal	Nominal	Standards Compliant	Nominal
Static array container type	No impact	No impact	No impact	No recommendation	C-style array
Indent style	K&R	K&R	K&R	K&R	K&R
Indent size	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	No impact	No impact	No impact	No impact	ert_code_-template.cgt
Code templates: Header file (*.h) template	No impact	No impact	No impact	No impact	ert_code_-template.cgt
Data templates: Source file (*.c) template	No impact	No impact	No impact	No impact	ert_code_-template.cgt
Data templates: Header file (*.h) template	No impact	No impact	No impact	No impact	ert_code_-template.cgt
File customization template	No impact	No impact	No impact	No impact	example_file_-process.tlc
Generate an example main program	No impact	No impact	No impact	No impact	On
Target operating system	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	No impact	Valid value	No impact	No impact	Auto
Data definition filename	No impact	Valid value	No impact	No impact	global.c
Data declaration	No impact	Valid value	No impact	No impact	Auto
Data declaration filename	No impact	Valid value	No impact	No impact	global.h
Use owner from data object for data definition placement	No impact	Valid value	No impact	No impact	off
#include file delimiter	No impact	Valid value	No impact	No impact	Auto
Signal display level	No impact	Valid integer	No impact	No impact	10
Parameter tune level	No impact	Valid integer	No impact	No impact	10
File packaging format	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
Specify custom data type names	No impact	On	No impact	No impact	Off
Replacement Name: double	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”

Embedded Coder Model Advisor Checks

Embedded Coder Checks

In this section...

"Check for blocks not recommended for C/C++ production code deployment" on page 21-3
 "Check configuration parameters for generation of inefficient saturation code" on page 21-4
 "Identify lookup table blocks that generate expensive out-of-range checking code" on page 21-5
 "Check output types of logic blocks" on page 21-6
 "Check the hardware implementation" on page 21-7
 "Identify questionable software environment specifications" on page 21-8
 "Identify questionable code instrumentation (data I/O)" on page 21-9
 "Identify blocks generating inefficient algorithms" on page 21-10
 "Check for unsupported block names" on page 21-11
 "Check usage of Assignment blocks" on page 21-12
 "Check for switch case expressions without a default case" on page 21-13
 "Check for missing error ports for AUTOSAR receiver interfaces" on page 21-14
 "Check bus object names that are used as bus element names" on page 21-15
 "Check configuration parameters for secure coding standards" on page 21-16
 "Check for blocks not recommended for secure coding standards" on page 21-18
 "Identify questionable subsystem settings" on page 21-19
 "Check usage of 'long long' data type" on page 21-20
 "Check for blocks not supported for row-major code generation" on page 21-20
 "Identify TLC S-Functions with unset array layout" on page 21-21
 "Identify blocks that generate expensive fixed-point and saturation code" on page 21-22
 "Check for missing const qualifiers in model functions" on page 21-24
 "Identify questionable fixed-point operations" on page 21-24
 "Identify blocks that generate expensive rounding code" on page 21-26
 "Check for bitwise operations on signed integers" on page 21-27
 "Check for recursive function calls" on page 21-28
 "Check for equality and inequality operations on floating-point values" on page 21-28
 "Check integer word lengths" on page 21-29
 "Check block names" on page 21-30
 "Check modeling style for component deployment" on page 21-31
 "Check signal interfaces" on page 21-32
 "Check Startup and Shutdown Event" on page 21-33
 "Check usage of partial data send" on page 21-34
 "Check configuration for component deployment" on page 21-34
 "Check for variant blocks that do not have a default choice" on page 21-35

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 information appear in tables of Simulink Block Support.

In some instances, this check flags blocks that are supported for code generation. For these blocks, you should review the footnote information that is provided in the support notes and adhere to the recommended action provided by the Model Advisor.

Following the recommendations of this check increases the likelihood of generating code that complies with the CERT C, CWE, and ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check™.

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.
The model or subsystem contains blocks that are supported but not recommended for production code generation.	Review the support notes and adhere to the recommended action provided by the Model Advisor.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyze content of library linked blocks.
- Analyze content in masked subsystems.
- Exclude blocks and charts if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “Use Products and Blocks Supported for Code Generation”
- “Model Advisor Exclusion Overview” (Simulink Check)

- Secure Coding Standards “Secure Coding”
- “Optimize Generated Code for Lookup Table Blocks”

Check configuration parameters for generation of inefficient saturation code

Check ID: `mathworks.codegen.EfficientTunableParamExpr`

Check model configuration for optimization settings that can improve code efficiency.

Description

This check identifies model configuration parameters that are recommended for C/C++ production code generation. For Embedded Coder, these model configuration parameters improve code efficiency.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The optimization suppresses code generation to guard against integer overflow for tunable parameter expression that you select. Select saturation code elimination.	If you have Embedded Coder and are using an ERT-based system target file, select Configuration Parameter Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values or set the parameter <code>EfficientTunableParamExpr</code> to on.
The optimization suppresses code generation that handles floating-point to integer conversion results for NaN values. Select conversion code elimination.	If you have Embedded Coder and are using an ERT-based system target file, select Configuration Parameter Remove code from floating-point to integer conversions with saturation that maps NaN to zero or set the parameter <code>EfficientMapNaN2IntZero</code> to on.

Action Results

Clicking **Modify Settings** configures model optimization settings can impact the efficiency of code generation. There are no safety concerns for:

- The **Remove code from tunable parameter expressions that saturates out-of-range values** parameter if your simulation contains the entire range of values for the parameters that are the terms of tunable expressions and Simulink does not generate a saturation warning.
- The **Remove code from floating-point to integer conversions with saturation that maps NaN to zero** parameter if your model simulation does not contain a NaN input value.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- Remove Code from Tunable Parameter Expressions That Saturate Out-of-Range Values
- “Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow”
- Remove code from floating-point to integer conversions with saturation that maps NaN to zero
- “Remove Code That Maps NaN to Integer Zero”

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
- Direct Lookup Table (n-D)
- 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, Direct 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.
- Cannot be used for AUTOSAR.
- 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”
- “Model Advisor Exclusion Overview” (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 <code>boolean</code> .	In the block dialog box, set Output data type to <code>boolean</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.

See Also

- “Model Advisor Exclusion Overview” (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 (ProdEndianness) • Production hardware signed integer division rounds to (ProdIntDivRoundTo) In the Configuration Parameters dialog box, specify the following parameters: • Byte ordering in test hardware (TargetEndianness) • 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

"Target Platform Device Customization"

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 > Identifiers pane, set the Maximum identifier length 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 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 > Identifiers pane are not set to recommended values.	In the Configuration Parameters dialog box, on the Code Generation > Identifiers pane, set the Generate scalar inlined parameters as on page 15-31 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-size signals, in the Configuration Parameters dialog box, on the Code Generation > Interface pane, clear Support: variable-size signals on page 14-16.
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. You might need to adjust the data types in your model after selecting the property. Note Starting in R2022a, you cannot disable the chart property Use strong data typing with Simulink I/O. For more information, see “Use strong data typing with Simulink I/O chart property has been removed” (Stateflow).

Limitations

A Stateflow license is required when using Stateflow charts.

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 pane, set the parameters to the recommended values.
Blocks generate assertion code.	In the Configuration Parameters dialog box, set Model Verification block enabling to Disable All on a block-by-block basis or globally.
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, 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

- “Model Advisor Exclusion Overview” (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 is set to Column-major for column-major code generation.	Disable the configuration parameter Use algorithms optimized for row-major array layout .

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 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 and MISRA C:2023 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Block names containing <code>/</code> were found in the model or subsystem.	Remove <code>/</code> 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:2023, Rule 3.1
- “MISRA C”
- “MISRA C Compliance Considerations”

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 and MISRA C:2023 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 in 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
- MISRA C:2023, Rule 9.1
- ISO/IEC TS 17961: 2013, uninitref
- CERT C, EXP33-C
- CWE, CWE-908
- “hisl_0029: Usage of Assignment blocks”
- “MISRA C”
- “MISRA C Compliance Considerations”

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 and MISRA C:2023 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
- MISRA C:2023, 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)
- “MISRA C”

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 and MISRA C:2023 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.
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: 2023, Directive 4.7
- “MISRA C”
- “Model Advisor Exclusion Overview” (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 and MISRA C:2023. 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 C:2023, Rule 5.6
- MISRA AC AGC, Rule 5.3
- “MISRA C”

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 or error and Support non-finite numbers is set to on.	When Support non-finite numbers is: • on, set Inf or NaN block output to warning. • off, set Inf or NaN block output to warning or error.
Configuration parameter Inf or NaN block output is set to None and Support non-finite numbers is set to off.	
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 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	

Condition	Recommended Action
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.x.	Set Code replacement library to None or AUTOSAR 4.x.
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 .
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 .
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 .
	Note Select only when Code Interface Packaging is set to Reusable Function.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Note When you click **Modify All** for models with a GRT-based target, the Model Advisor does not update the **System target file** configuration parameter to an ERT-based system.

Parameter subchecks depend on the results of the parameter noted with **D** in the results table. When the result is *D-Warning*, the **Current Value** column in the results table states *Prerequisite constraint not met* for the subchecks. After you change the parameter, rerun the check.

Note Some subchecks are specific to configuration parameters for ERT-based systems. These parameters are not updated when you click **Modify All** unless you change the model to an ERT-based system.

See Also

“MISRA C”

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

- “Model Advisor Exclusion Overview” (Simulink Check)
- “MISRA C”

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
- “Model Advisor Exclusion Overview” (Simulink Check)

Check usage of 'long long' data type

Check ID: `mathworks.codegen.EnableLongLong`

Check for usage of `long long` data type when expensive multi-word types are detected.

Description

This check flags when your Simulink model contains signal or port data types with word lengths larger than the `long` data type and the configuration parameter **Support long long** (`ProdLongLongMode` or `TargetLongLongMode`) is not enabled.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model contains signals or port data types with word lengths larger than the <code>long</code> data type, the Support long long parameter is not enabled, and your compiler supports the C <code>long long</code> data type.	Enable the Support long long parameter.

Capabilities and Limitations

- This check flags only explicit usages of data types with word lengths larger than the `long` data type in signals and ports. Implicit usage, such as the output of an operation, are not flagged by this check.

Edit-Time Checking

This check is supported by edit-time checking. Edit-time checking requires a Simulink Check license.

See Also

- “Support long long” (`ProdLongLongMode`)
- Support long long (`TargetLongLongMode`)

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 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** 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** 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 <code>SSArrayLayout</code> property to Column-major.
The configuration parameter Array layout is set to Row-major for row-major code generation.	Set the <code>SSArrayLayout</code> 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.
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.

Conditions	Recommended Action
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 on page 10-65 parameter in the Configuration Parameters dialog box.
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.
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)
- “Model Advisor Exclusion Overview” (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 and MISRA C:2023 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:2023, Rule 8.13
- “MISRA C”

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 Lookup Table Optimizer or FunctionApproximation.Problem.
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 Lookup Table Optimizer or FunctionApproximation.Problem.
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 Lookup Table Optimizer or FunctionApproximation.Problem.
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 Lookup Table Optimizer or FunctionApproximation.Problem.
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).

Conditions	Recommended Action
Blocks require cumbersome multiplication.	Restrict multiplication operations: • So the product integer size is not larger than the target integer size. • To the recommended size.
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)
- “Model Advisor Exclusion Overview” (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)
- “Model Advisor Exclusion Overview” (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 and MISRA C:2023 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.

Note Stateflow license is needed if the model contains Stateflow elements.

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

- Check operates at the generated code level rather than the model level. Consequently, it might not flag blocks if the generated code does not exhibit insecure input validation, even if the underlying model could potentially produce such issues. To overcome this limitation, use the Check usage of bit operation blocks (Simulink Check) check, which is designed to detect insecure input validation at the model level.

See Also

- MISRA C:2012, Rule 10.1

- MISRA C:2023, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “hisl_0060: Configuration parameters that improve MISRA C compliance”
- “MISRA C Compliance Considerations”
- “MISRA C”

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 and MISRA C:2023 compliant code for embedded applications. The check flags charts that have recursive function calls.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
Chart has a recursive function call.	Remove recursive function call.

See Also

- MISRA C:2012, Rule 17.2
- MISRA C:2023, Rule 17.2
- “Resolve Unintended Recursive Behavior” (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 and MISRA C:2023 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.

Note Stateflow license is needed if the model contains Stateflow elements.

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
- MISRA C:2023, Dir 1.1
- CERT C, FLP02-C
- CWE, CWE-1077
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “MISRA C”

Check integer word lengths

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 and MISRA C:2023 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.

Note Stateflow license is needed if the model contains Stateflow elements.

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
- MISRA C:2023, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “MISRA C”
- “Model Advisor Exclusion Overview” (Simulink Check)

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

- “Blocks Supported for Code Generation”

Check modeling style for component deployment

Check ID: mathworks.codegen.cgsl_0401

Description

Use this check to verify that the modeling style is compatible for deploying a component model that is configured with a service interface.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Modeling style is not compatible with requirements for deploying a component model configured with a service interface.	Update the model to use the export-function or rate-based modeling style. See “Periodic and Aperiodic Function Interfaces” for more information.

Capabilities and Limitations

There are no specified capabilities or limitations for this Model Advisor check.

Version History

Introduced in R2022b

See Also

“cgs_l_0401: Modeling styles for component deployment”

“Periodic and Aperiodic Function Interfaces”

“Create a Service Interface Configuration”

“Export-Function Models Overview”

“Create Export-Function Model”

“Schedule Components to Avoid Data Dependency Issues”

“Create Rate-Based Model”

Check signal interfaces

Check ID: `mathworks.codegen.cgs_l_0402`

Description

Use this check to verify that signal interfaces are used in a way that supports component model deployment.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model uses In Bus Element and Inport blocks or Out Bus Element and Outport blocks.	Update the model to use In Bus Element and Out Bus Element blocks or Inport and Outport blocks.
Incorrect setting for In Bus Element and Out Bus Element block parameter Bus virtuality .	Set block parameter Bus virtuality to nonvirtual .
Incorrect settings for Outport and Inport blocks.	Specify that the output bus is nonvirtual in the parent model by selecting block parameter Output as nonvirtual bus in parent model . Specify that the output for a top-level Inport block used to load bus data is nonvirtual by selecting Inport block parameter Output as nonvirtual bus .
Different sample times are used for the nonvirtual bus.	Update signals in the nonvirtual bus to have the same sample time.

Capabilities and Limitations

There are no specified capabilities or limitations for this Model Advisor check.

Version History

Introduced in R2022b

See Also

“cgsl_0402: Signal interfaces for component deployment”

“Modify Sample Times for Nonvirtual Buses”

“Specify Sample Time”

“View Sample Time Information”

Check Startup and Shutdown Event

Check ID: `mathworks.codegen.cgsl_0404`

Description

The check verifies that the model includes Initialize Function and Terminate Function blocks for modeling startup and shutdown behavior.

This check requires Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model does not use an Initialize Function block for modeling startup behavior.	Add an Initialize Function block to the component (top-level) model.
The model does not include a Terminate Function block for modeling shutdown behavior.	Use one of these options for shutdown events: - Add a Terminate Function block to the component (top-level) model. - Clear model configuration parameter Terminate function required.

Capabilities and Limitations

- Runs on export function models only.
- Run on the model root level.
- Function-call subsystems are allowed as function components for export-function modeling.
- Does not run on library models.
- Analyzes content of library-linked blocks.
- Analyzes content in masked subsystems.

Version History

Introduced in R2023a

See Also

“cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment”

“Startup, Reset, and Shutdown Function Interfaces”

“Periodic and Aperiodic Function Interfaces”

Check usage of partial data send

Check ID: `mathworks.codegen.cgsl_0408`

Description

The check verifies that block usage supports sending partial data.

This check requires Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model does not adhere to the export function modeling guideline	Update the model to use the export-function modeling style
The source for a Merge block is an Assignment block that is not inside of a Function-Call Subsystem.	Update the model so the source for <u>all</u> Merge blocks is an Assignment block inside a function-call subsystem.
The destination for a Merge block is not a virtual root-level Outport block.	Update the model so that the destination of the Merge block is a root-level Outport block inside of a function-call subsystem. Verify that Outport block parameter Ensure output is virtual is selected.
The Merge block is not configured for Direct Access data communication method.	Use the Direct Access data communication method for root-level Outport blocks that are connected to Merge blocks.

Capabilities and Limitations

- Run on the model root level.
- Analyzes content of library-linked blocks.
- Analyzes content in masked subsystems.

Version History

Introduced in R2023a

See Also

“cgsl_0408: Partial data send for component deployment”

“cgsl_0401: Modeling styles for component deployment”

“Data Communication Methods”

Check configuration for component deployment

Check ID: mathworks.codegen.cgsl_0414

Description

The check verifies that the component model is configured using a service interface.

This check requires Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Model is not linked to a shared Embedded Coder dictionary that is configured for service interface	Use configuration parameter Code Generation > Shared coder dictionary to select a shared dictionary that is configured as a service interface. To set up a new shared dictionary, in the Embedded Coder app, select Code Interface > Set up shared Embedded Coder Dictionary . Use the set up dialog box to create and link the model to a new dictionary and, when prompted, select Service Interface .
The deployment type of the top-level model is subcomponent.	In the Embedded Coder app, use the Deployment Type field to change the deployment type of the top model to Component .
The deployment type of the reference model is component.	In the Embedded Coder app, use the Deployment Type field to change the deployment type of the reference model to Subcomponent .

Capabilities and Limitations

- Does not work with protected models.

Version History

Introduced in R2022b

See Also

"cgsl_0414: Configure service interface for component model"

"Create a Service Interface Configuration"

Embedded Coder Dictionary

"Develop Deployable Code"

Check for variant blocks that do not have a default choice

Check ID: mathworks.misra.DefaultChoiceVariantChecks

Identify variant blocks with startup variant activation time that do not have a default choice when the **Casting modes** is set to **Standards Compliant**.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 and MISRA C:2023 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The variant blocks do not contain default choices but the Casting modes is set to Standards Compliant .	Add a default choice as one of the choices for variant blocks.

Capabilities and Limitations

- Analyzes content of variant blocks.

Edit-Time Checking

This check is supported by edit-time checking.

Check for blocks not recommended for MISRA C:2023

mathworks.misra.BlkSupport

Dependencies: Embedded Coder and Simulink Check

Usage: C/C++ Code Generation and High-Integrity System Modeling

Guideline: hisl_0020: Blocks not recommended for MISRA C compliance

Description

Use this check to identify blocks that are not supported or recommended for MISRA C:2012 and MISRA C:2023 compliant code generation. Following the recommendations of this check increases the likelihood of generating MISRA C:2012 and MISRA C:2023 compliant code for embedded applications.

Recommended Actions and Results

Review the violations that are flagged by the check and the recommended action for fixing the issue. After applying the changes, save the model and rerun the check analysis.

Modeling Condition	Recommended Action
These lookup table blocks that use cubic spline interpolation or extrapolation methods were found in the model or subsystem: 1-D Lookup Table 2-D Lookup Table - n-D Lookup Table	Consider other interpolation and extrapolation methods for the lookup table blocks.
These deprecated lookup table blocks were found in the model or subsystem: - 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.

Modeling Condition	Recommended Action
These blocks were found in the model or subsystem: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing these blocks with blocks recommended for production.

Capabilities and Limitations

- Runs on library models
- Does not require model compilation
- Supports edit-time checking
- Exclude blocks and Stateflow charts from this check if you have a Simulink Check license. See “Model Advisor Exclusion Overview” (Simulink Check).

Version History

Introduced in R2017b

See Also

Topics

“Check Your Model Using the Model Advisor”
 “Address Model Check Results”
 “Save and View Model Advisor Check Reports”
 “Improve Simulation Performance Using Performance Advisor”
 “Upgrade Models Using Upgrade Advisor”
 “Use Model Advisor Configuration Editor to Customize Model Advisor” (Simulink Check)
 “Exclude Blocks from Model Advisor Check Analysis” (Simulink Check)
 “Justify Model Advisor Violations from Check Analysis” (Simulink Check)
 “Check Model Compliance Using Edit-Time Checking” (Simulink Check)
 “Check Model Compliance by Using the Model Advisor” (Simulink Check)
 “Model Advisor Checks for MISRA C Coding Standards” (Simulink Check)
 “MISRA C”
 “MISRA C Compliance Considerations”

Check configuration parameters for MISRA C:2023

mathworks.misra.CodeGenSettings

Dependencies: Embedded Coder and Simulink Check

Usage: C/C++ Code Generation and High-Integrity System Modeling

Guideline: hisl_0060: Configuration parameters that improve MISRA C compliance

Description

Use this check to identify configuration parameters that can impact MISRA C:2012 and MISRA C:2023 compliant code generation. Following the recommendations of this check increases the likelihood of generating MISRA C:2012 and MISRA C:2023 compliant code for embedded applications.

Recommended Actions and Results

Review the violations that are flagged by the check and the recommended action for fixing the issue. After applying the changes, save the model and rerun the check analysis.

You can use the **Fix** button to allow the Model Advisor to fix all flagged violations. For this check, the Model Advisor changes the parameter values to the recommended values.

Note For models with a GRT-based target, the Model Advisor does not update the **System target file** configuration parameter to an ERT-based system.

The tables list recommended parameter settings based on their location in the Configuration Parameters dialog box.

Math and Data Types

Modeling Condition	Recommended Action
Model configuration parameter Use division for fixed-point net slope computation is not set to On or Use division for reciprocals of integers only.	Set model configuration parameter Use division for fixed-point net slope computation to On or Use division for reciprocals of integers only.

Modeling Condition	Recommended Action
Model configuration parameter Inf or NaN block output is set to None or error and Support: non-finite numbers is set to on.	When Support: non-finite numbers is set to on, set Inf or NaN block output to warning.
Model configuration parameter Inf or NaN block output is set to None and Support: non-finite numbers is set to off.	When Support: non-finite numbers is set to off, set Inf or NaN block output to warning or error.
Model configuration parameter Model Verification block enabling is set to Use local settings or Enable All.	Set model configuration parameter Model Verification block enabling to Disable All.
Model configuration parameter Undirected event broadcasts is set to none or warning.	Set model configuration parameter Undirected event broadcasts to error.
Model configuration parameter Wrap on overflow is set to None.	Set model configuration parameter Wrap on overflow to warning or error.

Hardware Implementation

Modeling Condition	Recommended Action
Model configuration parameter Signed integer division rounds to is set to Undefined.	Set model configuration parameter Signed integer division rounds to to Zero or Floor.
Model configuration parameter Shift right on a signed integer as arithmetic shift is selected.	Clear model configuration parameter Shift right on a signed integer as arithmetic shift .

Simulation Target

Modeling Condition	Recommended Action
Model configuration parameter Compile-time recursion limit for MATLAB functions is set to a value other than 0.	Set model configuration parameter Compile-time recursion limit for MATLAB functions to 0.
Model configuration parameter Dynamic memory allocation in MATLAB functions is selected.	Clear model configuration parameter Dynamic memory allocation in MATLAB functions .
Model configuration parameter Enable run-time recursion for MATLAB functions is selected.	Clear model configuration parameter Enable run-time recursion for MATLAB functions .

Code Generation

Modeling Condition	Recommended Action
Model configuration parameter Bitfield declarator type specifier is set to <code>uchar_T</code> 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 model configuration parameter Bitfield declarator type specifier to <code>uint_T</code> .
Model configuration parameter Casting modes is not set to <code>Standards Compliant</code> .	Set model configuration parameter Casting modes to <code>Standards Compliant</code> .
Model configuration parameter Code replacement library is not set to <code>None</code> or <code>AUTOSAR 4.x</code> .	Set model configuration parameter Code replacement library to <code>None</code> or <code>AUTOSAR 4.x</code> .
Model configuration parameter External mode is selected.	Clear model configuration parameter External mode .
Model configuration parameter Generate shared constants is selected.	Clear model configuration parameter Generate shared constants . For more information, see <code>GenerateSharedConstants</code> in “Model Configuration Parameters: Code Generation Interface”.
Model configuration parameter Include comments is cleared.	Select model configuration parameter Include comments .
Model configuration parameter MAT-file logging is selected.	Clear model configuration parameter MAT-file logging .
For ERT-based target systems, configuration parameter MATLAB user comments is cleared.	Select model configuration parameter MATLAB user comments .
A value for configuration parameter Maximum identifier length is not provided.	Set the value of model configuration parameter Maximum identifier length based on the limits for your implementation. The default value is 31.
Model configuration parameter Parentheses level is not set to <code>Standards (Parentheses for Standards Compliance)</code> or <code>Maximum (Specify precedence with parentheses)</code> .	Set model configuration parameter Parentheses level to <code>Standards (Parentheses for Standards Compliance)</code> or <code>Maximum (Specify precedence with parentheses)</code> .
For ERT-based target systems, configuration parameter Preserve static keyword in function declarations is cleared when File packaging format is set to <code>Compact</code> or <code>Compact (with separate data file)</code> .	Select model configuration parameter Preserve static keyword in function declarations .

Modeling Condition	Recommended Action
Model configuration parameter Replace multiplications by powers of two with signed bitwise shifts is selected.	Clear model configuration parameter Replace multiplications by powers of two with signed bitwise shifts .
Model configuration parameter Shared code placement is set to Auto.	Set model configuration parameter Shared code placement to Shared location.
For ERT-based target systems, configuration parameter Support: continuous time is selected.	Clear model configuration parameter Support: continuous time .
For ERT-based target systems, configuration parameter Support non-inlined S-functions is selected.	Clear model configuration parameter Support non-inlined S-functions .
Model configuration parameter Suppress generation of default cases for switch statements if unreachable is selected.	Clear model configuration parameter Suppress generation of default cases for switch statements if unreachable .
Model configuration parameter System-generated identifiers is set to Classic.	Set model configuration parameter System-generated identifiers to Shortened.
Model configuration parameter System target file is set to a GRT-based target.	Set model configuration parameter System target file to an ERT-based target.
Model configuration parameter Use dynamic memory allocation for model initialization is selected when Code Interface Packaging is set to Reusable Function.	Clear model configuration parameter Use dynamic memory allocation for model initialization .

Parameter subchecks depend on the results of the parameter noted with **D** in the results table. When the result status is D-Warning, the **Current Value** column in the results table reads `Prerequisite constraint not met` for the subchecks. After you change the parameter, rerun the check.

Note Some subchecks are specific to configuration parameters for ERT-based systems. These parameters are not updated when you use the **Fix** button unless you change the model to an ERT-based system.

Capabilities and Limitations

- Displays the settings of the following parameters as additional information in the check results:
 - Pack Boolean data into bitfields**
 - Code interface packaging (component)**
 - File packaging format**
 - Support: non-finite numbers**

- Does not review referenced models
- Does not require model compilation

Version History

Introduced in R2017b

See Also

Topics

“Check Your Model Using the Model Advisor”
“Address Model Check Results”
“Save and View Model Advisor Check Reports”
“Improve Simulation Performance Using Performance Advisor”
“Upgrade Models Using Upgrade Advisor”
“Use Model Advisor Configuration Editor to Customize Model Advisor” (Simulink Check)
“Exclude Blocks from Model Advisor Check Analysis” (Simulink Check)
“Justify Model Advisor Violations from Check Analysis” (Simulink Check)
“Check Model Compliance Using Edit-Time Checking” (Simulink Check)
“Check Model Compliance by Using the Model Advisor” (Simulink Check)
“Model Advisor Checks for MISRA C Coding Standards” (Simulink Check)
“MISRA C”
“MISRA C Compliance Considerations”

Simulink Coder Tools

Code Mappings Editor — C

Associate model elements with code interface definitions

Description

The Code Mappings editor is a graphical interface where you associate model elements with code interface definitions. When you generate code from the model, the code generator applies the specified definitions to the code it generates from the associated model elements. Each model in a model reference hierarchy has its own code mappings.

Definitions can include:

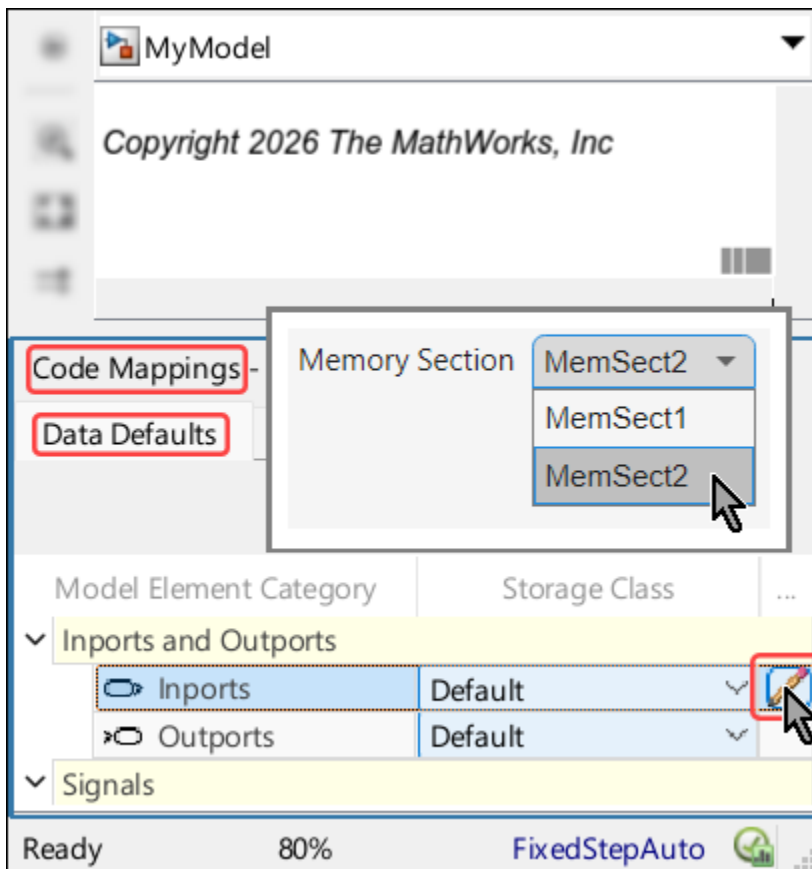
- **Storage Classes** — a storage class is a set of specifications for generating code from model data elements. It defines properties such as appearance and data types.
- **Function Customization Templates** — a function customization template is a set of specifications for generating entry-point functions. It defines properties such as naming rules for function names and argument related identifiers.
- **Memory Sections** — a memory section is a location in the computer memory, designated to contain specific code elements.
- **Service Interfaces** — a service interface is set of specifications that defines a function prototype and a communication method. Each such interface corresponds to a function that is available to be invoked by the generated model code on the target platform. Tuning and measurement services are used by the generated model code to tune and measure model elements, respectively. Sender, receiver, and data transfer services are used for model elements communication. Timer services are used to obtain real-time information about the target platform clock. To learn more about service interfaces, see “Service Interfaces”.

The system target file you specify for your model, as well as other settings, determine which code interface definition types are available for configuration. The rest are determined automatically by the code generator.

With models configured to use the GRT system target file you can only configure storage class definitions. For each data element category, associate a default storage class to use throughout the model. Then, override the defaults, as needed, by specifying storage classes for individual data elements.

Models configured to use the ERT system target file can be configured with either data interface configuration or with service interface configuration.

With ERT models configured with data interface configuration, associate each category of data elements with a default storage class to use throughout the model, and associate each category of entry-point functions with a default function customization template to use throughout the model. Then, override the default definitions, as needed, by specifying storage classes for individual data elements and by specifying function customization templates for individual entry-point functions. In some configurations, you can explicitly specify a memory section on top of the specified storage class or function customization template. This is possible when the definition of the specified storage class or function customization template does not include a memory section. For example, when you specify `Default` as the default storage class for the model **Inports** category, as shown in this screenshot:



A service interfaces dictionary is an Embedded Coder dictionary that contains service interface definitions, sorted by types. For each service type, one of the definitions is selected in the dictionary as the default service for the type. In addition, the dictionary contains definitions of storage classes, memory sections, and function customization templates. The dictionary specifies the default function customization template for each entry-point function category.

When an ERT model is linked to a shared Embedded Coder dictionary that defines service code interfaces, it means that the model is configured with service interfaces configuration. By default, the code generated from such models adheres to the default definitions, as specified in the dictionary. You can, however, use the Code Mappings editor to override the default code interface definitions, by associating individual entry-point functions with function customization templates from the dictionary and by associating individual model elements with services from the dictionary.

To configure model elements for code generation, use the Code Mappings editor tabs:

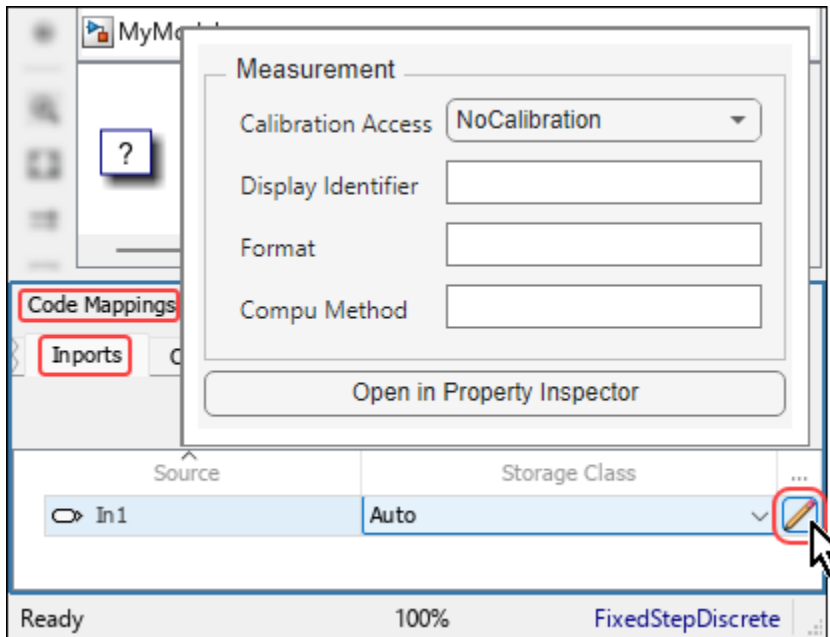
- **Data Defaults** (only applicable for GRT models and ERT models with data interface configuration)
- **Function Defaults** (only applicable for ERT models with data interface configuration)
- **Functions** (only applicable for ERT models)
- **Inports**
- **Outports**
- **Data Transfers** (only applicable for ERT models with service interface configuration)
- **Parameters**

- **Data Stores**
- **Signals/States**

Depending on the type of element and the specified value, there could be additional configurable code mapping properties you can set. To configure the additional properties, select the pencil icon



at the end of the row to open a dialog box. For example, this screenshot shows the dialog box that opens when you click the pencil icon for a root-level inport with **Auto** storage class.



Signals, by default, are not added to the code mappings of the model. To configure a signal for code generation, add the signal to the model code mappings. To learn more about adding signals, see “Add Signals to Model Code Mappings”. To learn more about removing signals, see “Remove Signals from Model Code Mappings”.

Open the Code Mappings Editor — C

To open the editor, do one of the following:

- In the model canvas of the Simulink editor, click the perspective control in the lower-right corner and select **Code**. Then, click the **Code Mappings** tab.
- Open the Simulink Coder app or the Embedded Coder app. Then, In the bottom left corner of the Simulink Editor window, click the **Code Mappings** tab.
- Open the Simulink Coder app or the Embedded Coder app. Then, on the **C Code** tab of the Simulink toolstrip, in the prepare section, select **Code Interface > Individual Element Code Mappings**.

Examples

Configure Code Generation for Root-Level Inports and Outports With Simulink Coder

Configure code generation for the root-level 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. To configure code generation for root-level In Bus Element blocks and Out Bus Element blocks use the same workflow as in the example.

Set Up Example Environment

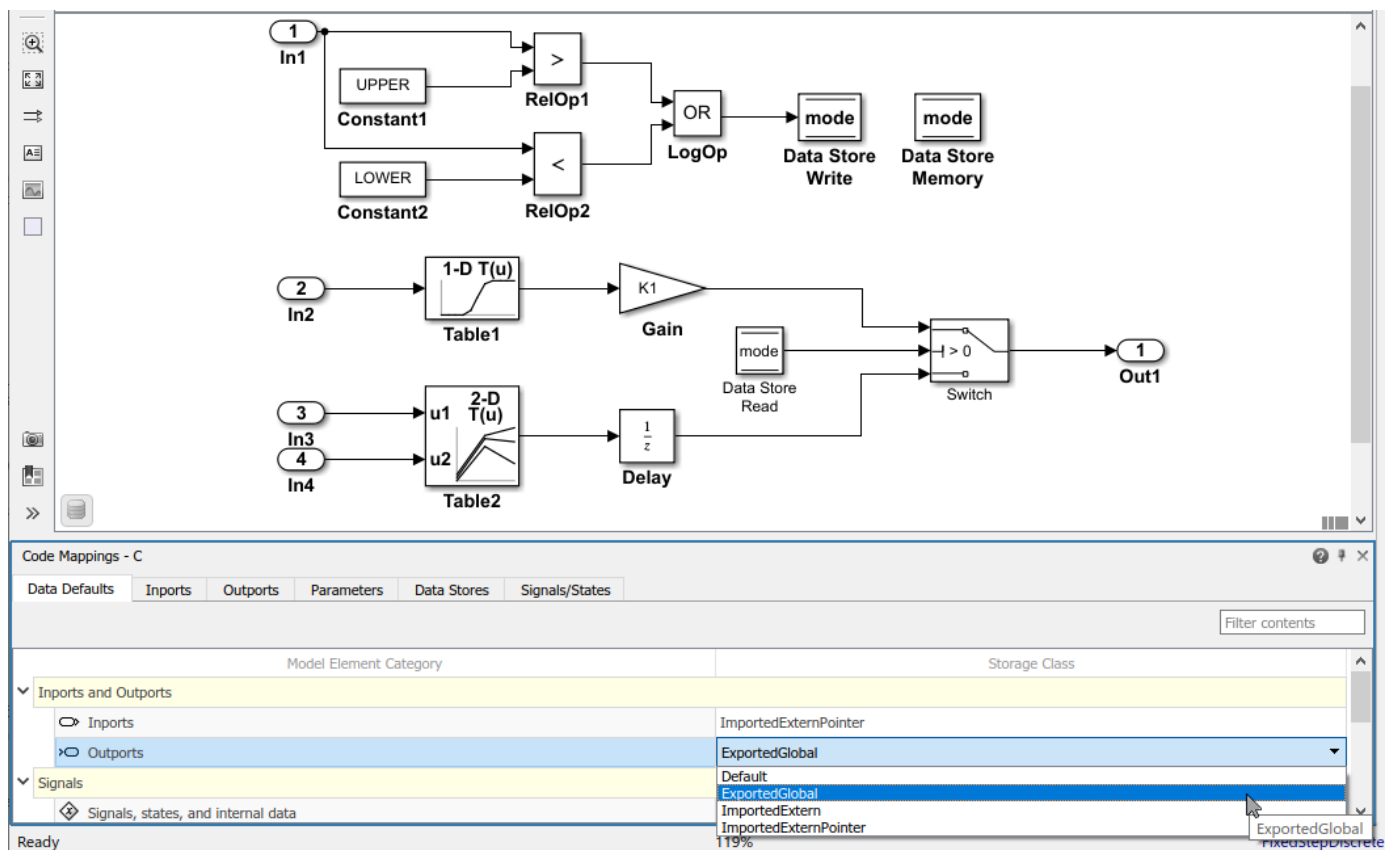
- 1 Open the model `ConfigurationRapidPrototypingInterface` by entering this command at the MATLAB Command Window:

```
openExample("ConfigurationRapidPrototypingInterface")
```
- 2 Open the Simulink Coder app. The **C Code** tab includes the Code Mappings editor.

Configure Default Mappings

Configure the code generator to declare and define global variables for inports and outports in generated files `ConfigurationRapidPrototypingInterface.h` and `ConfigurationRapidPrototypingInterface.c`.

- 1 In the **C Code** tab, select **Code Interface > Default Code Mappings**.
- 2 In the **Data Defaults** tab, under **Inports and Outports**, select the row for **Inports**. Then, set the storage class to `ImportedExternPointer`. Set the storage class for **Outports** to `ExportedGlobal`. The editor updates the default storage class setting for the two selected data element categories.



Configure Individual Inports and Outports for Default Configuration

- 1 In the Code Mappings editor, click the **Inports** tab. When the storage class is set to **Auto**, the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the model default configuration.
- 2 Force the code generator to use the default configuration for inports, which is storage class **ImportedExternPointer**. Press the **Ctrl** key and select the inports. For one of the selected inports, set the storage class to **Model default: ImportedExternPointer**. The editor updates the storage class setting for the selected inports.
- 3 Force the code generator to use storage class **ExportedGlobal** for the model root-level outport. Click the **Outports** tab. Select the row for Out2. Then, set the storage class to **Model default: ExportedGlobal**.

Configure Individual Data Elements

To configure properties for individual data elements, for example, if you need to override default configuration settings, use the tabs for the different data element types. For this example, override the default storage class setting for Inport block In1.

By default, the code generator names inport and outport variables based on the Inport or Outport block name in the model. When you configure data elements with a storage class setting other than **Auto**, you can override that default setting for individual elements by setting storage class property **Identifier**. This property enables you to specify an identifier for the code without modifying the model design.

For this example, set **Identifier** for the Inport and Outport blocks.

- 1 In the Code Mappings editor, click the **Inports** tab.
- 2 For In1, set the storage class to ImportedExtern.
- 3 For each inport, select the row. Then, click the pencil icon  and set the **Identifier** property as follows:
 - Set In1 to input1.
 - Set In2 to input2.
 - Set In3 to input3.
 - Set In4 to input4.
- 4 Click **Outputs**.
- 5 Select output Out1. Click the pencil icon  and set the **Identifier** property to output.

Generate and Verify Code

Generate code and verify that the code generated for the Inport and Outport blocks appears as you expect. For example:

- ConfigurationRapidPrototypingInterface_private.h includes these declarations:

```
/* Imported (extern) block signals */
extern real_T input1;                /* '<Root>/In1' */

/* Exported data declaration */

/* Data with Imported storage (pointer) */
extern real_T *input2;               /* '<Root>/In2' */
extern real_T *input3;               /* '<Root>/In3' */
extern real_T *input4;               /* '<Root>/In4' */
```

- ConfigurationRapidPrototypingInterface.h includes these declarations.

```
/* Data with Exported storage */
extern real_T output;               /* '<Root>/Out1' */
```

- This code fragment shows the variable that represents In1, input1, being used in the generated entry-point step function.

```
/* Model step function */
void ConfigurationRapidPrototypingInterface_step(void)
{
    /* DataStoreWrite: '<Root>/Data Store Write' incorporates:
     * Constant: '<Root>/Constant1'
     * Constant: '<Root>/Constant2'
     * Inport: '<Root>/In1'
     * Logic: '<Root>/LogOp'
     * RelationalOperator: '<Root>/RelOp1'
     * RelationalOperator: '<Root>/RelOp2'
     */
    mode = ((input1 > 10.0) || (input1 < -10.0));
    .
    .
    .
}
```

Configure Code Generation for Root-Level Inports and Outports With Embedded Coder Using a Data Code Interface

Configure code generation for the root-level 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. To configure code generation for root-level In Bus Element and Out Bus Element blocks use the same workflow as in the example.

Set Up Example Environment

- 1 Open the model `RollAxisAutopilot` by entering this command at the MATLAB Command Window:

```
openExample("RollAxisAutopilot")
```

The current working directory contains these external code files.

- `roll_input_data.c`
- `roll_input_data.h`
- `roll_heading_mode.c`
- `roll_heading_mode.h`

- 2 Open the **Embedded Coder** app.

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`.
 - Configure names that the code generator uses for variables it produces in the code for Inport blocks.
- 1 In the **C Code** tab, select **Code Interface > Default Code Mappings**.
 - 2 In the **Data Defaults** tab, under **Inports and Outports**, select the row for **Inports**. Then, set the storage class to `ImportFromFile`.
 - 3 Click the pencil icon  and set **Header File** to `roll_input_data.h`.
 - 4 Set the storage class for model element category **Outports** to `ExportToFile`.
 - 5 Set **Header File** to `roll_output_data.h` and **Definition File** to `roll_output_data.c`.

Configure Individual Inports for Default Configuration

- 1 In the Code Mappings editor, click the **Inports** tab. The storage class for each inport is set to `Auto`, which means that the code generator might eliminate or change the representation of relevant code for optimization purposes. If optimizations are not possible, the code generator applies the default configuration for inports.
- 2 Force the code generator to use the default configuration for inports, storage class `ImportFromFile` with external header file `roll_input_data.h`. Press the **Ctrl** key and select the inports. For one of the selected inports, set the storage class to `Model default: ImportFromFile`. The editor updates the storage class setting for the selected inports.

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 In the Code Mappings editor, click the **Inports** tab.

- 2 Select the HDG_Mode row.
 - 3 Set **Storage Class** to ImportFromFile.
 - 4 Click the pencil icon  and set **Header File** to roll_heading_mode.h.
 - 5 Configure the code generator to produce variable names in the code for the Inport blocks that match the variable names in external files roll_input_data.h and roll_heading_mode.h.
- On the **Inports** tab, select each Inport block, click the pencil icon  and set **Identifier** to the block name. When the storage class is set to a value other than **Auto**, you can specify a value for the **Identifier** storage class property. If you leave the **Identifier** property empty, the code generator uses the name of the block or signal associated with the modeling element.

Include External Source Files In Code Generation and Build Process

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.

Save the model.

Generate and Verify Code

Generate code and verify that the code generated for the Inport and Outport blocks appears as you expect.

- RollAxisAutopilot.h includes these header files associated with storage classes:

```
#include "roll_output_data.h"
#include "roll_input_data.h"
#include "roll_heading_mode.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 With Embedded Coder Using a Data Code Interface

By default, the code generator uses the identifier naming rule `$R$N` to name entry-point functions. `$R` is the name of the top model. `$N` is the name of the function, for example, `initialize`, `step`, and `terminate`. To integrate generated 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

- 1 Open the model `MultirateMultitasking` by entering this command at the MATLAB Command Window:

```
openExample("MultirateMultitasking")
```
- 2 Open the Embedded Coder app. The **C Code** tab opens on the Simulink Toolstrip. You can open the Code Mappings editor pane by selecting **Code Mappings - Component Interface** near the status bar.

Define Function Naming Rule

Create a function customization template that defines the naming rule `myproj_$R$N`.

- 1 Open the Embedded Coder Dictionary. In the **C Code** tab, select **Code Interface > Embedded Coder Dictionary (Model)**.
- 2 In the **Data Interface** pane, select **Function Customization Templates**.
- 3 In the center pane, click **Create**.
- 4 In the right pane, specify **Name** as `myproj_FunctionTemplate`.
- 5 Specify **Function Naming Rule** as `myproj_$R$N`.
- 6 Close the Embedded Coder Dictionary.

Configure Default Mappings

- 1 In the **C Code** tab, select **Code Interface > Default Code Mappings**.
- 2 Click the **Function Defaults** tab.
- 3 For the **Initialize/Terminate** and **Execution** function categories, change the default function customization template from `Default` to `myproj_FunctionTemplate`.

The screenshot shows the Simulink environment with the 'MultirateMultitasking' model open. The model diagram includes two input ports, 'In1_1s' and 'In2_2s', an 'Integrator' block, two state space blocks 'SS1' and 'SS2', and a summing junction. The 'Code Mappings - Component Interface' window is open at the bottom, showing the 'Function Defaults' tab. It lists three functions: 'Initialize/Terminate', 'Execution', and 'Shared utility'. The 'Execution' function is selected, and its 'Function Customization Template' is set to 'myproj_FunctionTemplate'.

Model Function Category	Function Customization Template
fx Initialize/Terminate	myproj_FunctionTemplate
fx Execution	Default
fx Shared utility	myproj_FunctionTemplate

4 Save the model.

Generate and Review Code

Generate code and verify the entry-point function names.

```
void myproj_MultirateMultitasking_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_MultirateMultitasking_step1(void) /* Sample time: [2.0s, 0.0s] */
{
    rtDW.RateTransition_Buffer0 = rtDW.Integrator_DSTATE;
    rtDW.Integrator_DSTATE += 2.0 * rtU.In2_2s;
}

void myproj_MultirateMultitasking_initialize(void)
{
    /* (no initialization code required) */
}
```

```
void myproj_MultirateMultitasking_terminate(void)
{
    /* (no terminate code required) */
}
```

Customize Individual Entry-Point Functions With Embedded Coder Using a Data Code Interface

For your model, you can customize the names of most entry-point functions and the arguments of execution functions, such as step functions and Simulink functions. This example shows how to customize the entry-point functions for the model RollAxisAutopilot.

Set Up the Environment

- 1 Open the model RollAxisAutopilot by entering this command at the MATLAB Command Window:

```
openExample("RollAxisAutopilot")
```

The current working directory contains these external code files.

- roll_input_data.c
- roll_input_data.h
- roll_heading_mode.c
- roll_heading_mode.h

- 2 Open the Embedded Coder app. The **C Code** tab opens on the Simulink Toolstrip. You can open the Code Mappings editor pane by selecting **Code Mappings - Component Interface** near the status bar.

Customize Entry-Point Functions

- 1 In the **C Code** tab, select **Code Interface > Individual Element Code Mappings**. The Code Mappings editor opens with the **Functions** tab selected.
- 2 Customize the name of the step (execution) function, with **Source Periodic:D1**. In the **Function Name** column, enter the name roll_run.
- 3 Customize arguments of the step function. Open the configuration dialog box for the step function by clicking the prototype hyperlink in the **Function Preview** column.
- 4 Select **Configure arguments for Step function prototype**.
- 5 To open a table that displays the default configurations for the arguments, click **Get Default**.
- 6 Customize the arguments:
 - 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 Qualifier** drop-down list, select Pointer. In the **C Identifier Name** field change the name to HDG_Mode_Ptr
- 7 Click **Apply** and verify that the function prototype reflects the changes.

Configure C Step Function Interface: RollAxisAutopilot

Configure the generated C function interface, including function name and arguments.

C function prototype:arg_Ail_Cmd = roll_run(Phi, Psi, Rate_FB, TAS, AP_Eng, * HDG_Mode_Ptr, HDG_Ref, Turn_Knob)

C Step Function Name:roll_run

☒ Configure arguments for Step function prototype

Get default(* invokes update diagram)

C return argumentAil_Cmd

Port Name	Port Type	C Type Qualifier	C Identifier Name
Rate_FB	Inport	Value	Rate_FB
TAS	Inport	Value	TAS
AP_Eng	Inport	Value	AP_Eng
HDG_Mode	Inport	Pointer	HDG_Mode_Ptr
HDG_Ref	Inport	Value	HDG_Ref
Turn_Knob	Inport	Value	Turn_Knob

Drag and drop rows to specify argument order

Validate(* invokes update diagram)

Press Validate button to get validation results.

OKCancelHelpApply

- 8 Validate the changes by clicking **Validate**.
- 9 Click **OK**.

Generate and Verify Code

- 1 Generate code.
- 2 Verify the updates in the generated C file RollAxisAutopilot.c. To find the updated step function (roll_run), use the **Search** field.
- 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_Ptr, real32_T HDG_Ref,
                 real32_T Turn_Knob)
```

Parameters

Data Defaults (only applicable for GRT models and ERT models with data interface configuration)

Model Element Category — Category of model data elements
character vector

Category of model data elements for which to apply the code interface definitions. This table summarizes the model element categories.

Model Element Category	Description	Programmatic Category Identifier
Inports	Root-level input ports of a model, such as Inport and In Bus Element blocks.	Inports
Outports	Root-level output ports of a model, such as Outport and Out Bus Element blocks.	Outports
Signals, states, and internal data	Data elements that are internal to the model, such as block output signals, discrete block states, data stores, and zero-crossing signals.	InternalData
Shared local data stores	Data Store Memory blocks that have 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.	SharedLocalDataStores
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. To view and configure these data stores in the Code Mappings editor, click the Refresh link to the right of the category name. Clicking this link updates the model diagram.	GlobalDataStores
Model parameters	Parameters that are defined within a model, such as parameters in the model workspace. Excludes model arguments.	ModelParameters
External parameters	Parameters that you define as objects in the base workspace or in a data dictionary. Multiple models in an application can use these parameters. To view and configure these parameters in the Code Mappings editor, click the Refresh link to the right of the category name. Clicking this link updates the model diagram.	ExternalParameters
Model parameter arguments (only applicable for ERT models)	Parameters in the model workspace that you configure as model arguments. These parameters are exposed at the model block to enable each model instance to provide its own value. To specify a parameter as a model argument, select the Model Data Editor > Parameters > Argument check box.	ModelParameterArguments
Constants (only applicable for ERT models)	Constant-value block output and parameters that could not be inlined. These values are stored in variables for one of the following reasons. - The value is an array larger than the loop unrolling threshold. - The value address is needed in the code.	Constants

To programmatically access the default value of a data elements category, use the functions `getDataDefault` and `setDataDefault` with the code mappings object of the model and the

programmatic category identifier. For example, to programmatically specify the default storage class of model inports as `ExportedGlobal`, use the programmatic identifier of inports, `Inports`:

```
codeMapObj = coder.mapping.api.get(simulinkModel);
setDataDefault(codeMapObj, "Inports", StorageClass="ExportedGlobal");
```

The Code Mappings editor presents valid storage class options for a given category. The options can include:

- Unspecified storage class (**Default**). The code generator places the code for the category of data elements in standard structures, such as `B_`, `ExtY_`, `ExtU_`, `DW_`, and `P_`. See “Data Structures in the Generated Code”.
- Relevant predefined storage classes, such as `ExportedGlobal`.
- Relevant storage classes in available packages, such as the storage class `ImportFromFile` (only applicable for ERT models).
- Storage class defined in an Embedded Coder dictionary (only applicable for ERT models).

Storage Class — Default storage class for category

`Default` (default) | `ExportedGlobal` | `ImportedExtern` | `ImportedExternPointer` | dictionary defined storage class (only for ERT models with data interface configuration)

Default storage class definitions to use for generating code from model elements of the category. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Memory Section (only applicable for ERT models with data interface configuration) —

Default memory section to place code generated from elements of the category

`None` (default) | dictionary defined memory section

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

When the specified storage class definition does not include a memory section, use the drop-down list to select a memory section to place the generated code in.

Function Defaults (only applicable for ERT models with data interface configuration)

Model Function Category — Category of entry-point functions

`Initialize/Terminate` | `Execution` | `Shared utility`

Category of entry-point functions for which to apply the code interface definitions. This table summarizes the entry-point function categories.

Model Function Category	Description	Programmatic Category Identifier
Initialize/Terminate	Entry-point functions for initialization and termination	InitializeTerminate
Execution	Entry-point functions for initiating execution and resets	Execution
Shared utility	Shared utility functions	SharedUtility

To programmatically access the default value of a function category, use the functions `getFunctionDefault` and `setFunctionDefault` with the code mappings object of the model and

the programmatic category identifier. For example, assume that you defined the function customization template FT_01 in the Embedded Coder dictionary of the model. To programmatically specify the default function customization template of the initialization and termination functions category as FT_01, use the programmatic identifier of initialization and termination functions, `InitializeTerminate`:

```
codeMapObj = coder.mapping.api.get(simulinkModel);  
setFunctionDefault(codeMapObj,"InitializeTerminate",FunctionCustomizationTemplate="FT_01")
```

Function Customization Template — Default function customization template for category
Default (default) | Dictionary default | dictionary defined function customization template

Default function customization template definitions to use for generating entry-point functions of the category.

Memory Section — Memory section to place the generated code in
None (default) | dictionary defined memory section

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

When you specify Default as the function customization template for the category, you can select a default memory section, from those defined in the dictionary, to place generated entry-point functions of the category.

Functions (only applicable for ERT models)

Source — Entry-point function identifier
character vector

The identifier of the of entry-point function. For more information, see “Types of Generated Entry-Point Functions”.

Function Customization Template — Function customization template for function
Default (default) | Model default | dictionary defined function customization template

Function customization template to use for generating the entry-point function code.

Function Name — Name of generated entry-point function
character vector

The name of the generated entry-point function.

Function Preview — Preview of function prototype
character vector

Preview of the generated entry-point function prototype. To verify a prototype, review the prototype preview. To open a dialog box where you can customize the prototype, click the preview hyperlink. For more information, see “Configure Default Settings for Functions”.

Memory Section (only applicable for ERT models with data interface configuration) —
Memory section to place the generated code in
None (default) | Model default | dictionary defined memory section

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Use the drop-down list to select a memory section for the generated entry-point function.

Timer Service (only applicable for ERT models with service interface configuration) — Name of timer service interface

Dictionary default (default) | dictionary defined timer service

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Use the drop-down list to select a timer service, from those defined in the dictionary, to use with the generated entry-point function. For more information, see “Configure Timer Service Interfaces”.

Inports

Source — Name of root-level inport

character vector

Identifies a root-level Inport block or an In Bus Element block (for example, InBus1.signal1) in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is **Auto**. If the storage class is **Auto**, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to **From signal object**: followed by the name of the storage class of the data object. If the storage class is not **Auto**, the data element assumes the configuration that you specify in the Code Mappings editor.

Storage Class — Storage class of generated inport

Default (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer | dictionary defined storage class (only for ERT models with data interface configuration)

Storage class definitions to use for generating code from the inport. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Identifier — Name of variable

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name for the variable that represents the inport in the generated code.

Calibration Access — Enable or disable calibration of the inport

NoCalibration (default) | Calibration

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Select **Calibration** for the inport to enable the calibration, or **NoCalibration** to view the value of the inport and disable the calibration.

Display Identifier — Display name of the inport

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Optional display name of the inport for the measurement purpose in the calibration tool, which is different from the inport name in the Simulink model.

Format — Display format of the inport value

%[length].[layout]

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in **Compu Method** of the inport.

Compu Method — Name of conversion method

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Receiver Service (only applicable for ERT models with service interface configuration) —

Name of receiver service interface

Dictionary default (default) | dictionary defined timer service

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Use the drop-down list to select a receiver service, from those defined in the dictionary, to use with the code generated from the inport. For more information, see “Configure Sender and Receiver Service Interfaces for Model Inports and Outports”.

Outputs**Source** — Name of root-level outport

character vector

Identifies a root-level Outport block or an Out Bus Element block (for example, OutBus1.signal1) in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is **Auto**. If the storage class is **Auto**, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to **From signal object:** followed by the name of the storage class of the data object. If the storage class is not **Auto**, the data element assumes the configuration that you specify in the Code Mappings editor.

Storage Class — Storage class of generated output

Default (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer | dictionary
defined storage class (only for ERT models with data interface configuration)

Storage class definitions to use for generating code from the output. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Identifier — Name of variable

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name for the variable that represents the output in the generated code.

Calibration Access — Enable or disable calibration of the output

NoCalibration (default) | Calibration

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Select **Calibration** for the output to enable the calibration, or **NoCalibration** to view the value of the output and disable the calibration.

Display Identifier — Display name of the output

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Optional display name of the output for the measurement purpose in the calibration tool, which is different from the output name in the Simulink model.

Format — Display format of the output value

%[length].[layout]

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Special display format to be specified for measurement in the calibration tool. This format specification overrides the display format specified in **Compu Method** of the output.

Compu Method — Name of conversion method

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Sender Service (only applicable for ERT models with service interface configuration) —

Name of sender service interface

Dictionary default (default) | dictionary defined timer service

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Use the drop-down list to select a sender service, from those defined in the dictionary, to use with the code generated from the output. For more information, see “Configure Sender and Receiver Service Interfaces for Model Imports and Outputs”.

Data Transfers (only applicable for ERT models with service interface configuration)**Source** — Name of signal line

character vector

Identifies a signal line that connects two blocks that result in callable entry-point functions in the generated code. To learn more about callable entry-point functions, see “Model Code Interfaces”.

Data Transfer Service — Name of data transfer service interface

character vector

Data transfer service interface to use for code generation, specified as one of the service interfaces defined in the Embedded Coder Dictionary of the model. Within the target environment, the callable entry-point functions transfer data between them by calling the specified data transfer service of the target platform. To use the default data transfer service of the dictionary, specify `Dictionary default`. To configure a data transfer service interface, you must attach an Embedded Coder Dictionary that defines the service interface configuration to the model. For more information, see “Configure Data Transfer Service Interfaces for Data Transfer Signals”.

Parameters**Source** — Name of model parameter argument, model parameter, or external parameter

character vector

Identifies a parameter in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-parameter-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is `Auto`. If the storage class is `Auto`, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to `From parameter object:` followed by the name of the storage class of the data object. If the storage class is not `Auto`, the data element assumes the configuration that you specify in the code mappings.

Types of parameter elements are listed in this table.

Type of Parameter Element	Description
Model parameter argument (only applicable for ERT models)	Block parameter in the model workspace that you configure as a model argument. The parameter is exposed at the model block to enable each model instance to provide its own value. To specify a parameter as a model argument, select the Model Data Editor > Parameters > Argument check box.

Type of Parameter Element	Description
Model parameter	Parameter that is defined within a model, such as a parameter in the model workspace. Excludes model arguments.
External parameter	Parameter that you define as an object in the base workspace or in a data dictionary. Multiple models in an application can use these parameters. This grouping of parameters appears in the editor only if the model uses such an element. To view and configure these parameters in the Code Mappings editor, click the Refresh link to the right of the category name. Clicking this link updates the model diagram.

Storage Class — Storage class of generated parameter

Default (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer | dictionary defined storage class (only for ERT models with data interface configuration)

Storage class definitions to use for generating code from the parameter. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Identifier — Name of variable

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name for the variable that represents the model parameter or model parameter argument in the generated code.

Calibration Access — Enable or disable calibration of the parameter

NoCalibration (default) | Calibration

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Select Calibration for the parameter to enable the calibration, or NoCalibration to view the value of the parameter and disable the calibration.

Display Identifier — Display name of parameter

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Optional display name of the parameter for the measurement purpose in the calibration tool, which is different from the parameter name in the Simulink model.

Format — Display format of the parameter value

%[length].[layout]

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in **Compu Method** of the parameter.

Compu Method — Name of conversion method
character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Parameter Tuning Service (only applicable for ERT models with service interface configuration) — Name of parameter tuning service interface
character vector

Name of a parameter tuning service interface defined in the Embedded Coder Dictionary. To use the dictionary default, specify **Dictionary default**.

To configure the parameter tuning service interface, you must attach an Embedded Coder Dictionary that defines a service interface configuration to the model. For more information, see “Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments”.

Data Stores

Source — Name of local data store, shared local data store, or global data store
character vector

Identifies a data store in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve-to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is **Auto**. If the storage class is **Auto**, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to **From signal object**: followed by the name of the storage class of the data object. If the storage class is not **Auto**, the data element assumes the configuration that you specify in the code mappings.

Types of data store elements are listed in this table.

Type of Data Store Element	Description
Local data store	Data store that is accessible from anywhere in the model hierarchy that is at or below the level at which you define the data store. You can define a local data store graphically in a model by including a Data Store Memory block or by creating a signal object (synthesized data store) in the model workspace.
Shared local data store	Data Store Memory block that has 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. This grouping of data stores appears in the editor only if such an element exists in the model.

Type of Data Store Element	Description
Global data store	Data store that is defined by a signal object in the base workspace or in a data dictionary. Multiple models in an application can use these data stores. These data stores are not configurable in the code mappings. After you click the refresh button, they appear in the Code Mappings editor in a read-only state for viewing purposes. This grouping of data stores appears in the editor only if the model uses such an element. To view and configure these data stores in the Code Mappings editor, click the Refresh link to the right of the category name. Clicking this link updates the model diagram.

Names of local and shared local data stores appear in the format *block-name: data-store-name*.

Depending on how the data store element is represented and configured in the model, local and shared local data stores can resolve to a signal object in the model workspace, based workspace, or a data dictionary. Global data stores resolve to a signal object in the base workspace or a data dictionary.

Storage Class — Code definition for data store

Default (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer | dictionary
defined storage class (only for ERT models with data interface configuration)

Storage class definitions to use for generating code from the data store. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Path — Path to data store in model

character vector

Link that you can click to highlight the data store in the model diagram.

Identifier — Name of variable

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name for the variable that represents the data store in the generated code.

Calibration Access — Enable or disable calibration of the data store

NoCalibration (default) | Calibration

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Select Calibration for the data store to enable the calibration, or NoCalibration to view the value of the data store and disable the calibration.

Display Identifier — Display name of the data store

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Optional display name of the data store for the measurement purpose in the calibration tool, which is different from the data store name in the Simulink model.

Format — Display format of the data store value

`%[length].[layout]`

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Special display format to be specified for measurement in the calibration tool. This format specification overrules the display format specified in `Compu Method` of the data store.

Compu Method — Name of conversion method

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Measurement Service (only applicable for ERT models with service interface configuration)

— Name of measurement service interface

Dictionary default (default) | Not measured | dictionary defined measurement service

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Use the drop-down list to specify that the signal or state should not be measured, or select a measurement service, from those defined in the dictionary, to use for measuring the data store. For more information, see “Configure Measurement Service Interfaces for Signals, States, and Data Stores”.

Signals/States

Source — Name of signal or state

character vector

Identifies a signal line or state in the model. If the element resolves to a data object, the Code Mappings editor displays a resolve -to-signal-object icon to the right of the source name and resolves the configuration based on whether the storage class setting for the element is `Auto`. If the storage class is `Auto`, the data element assumes the code configuration that the data object specifies. The editor changes the display text in the **Storage Class** column to `From signal object:` followed by the name of the storage class of the data object. If the storage class is not `Auto`, the data element assumes the configuration that you specify in the Code Mappings editor.

The Code Mappings editor lists:

- Named signals and states by using the data element name

- Unnamed signals by using the format *source-block: port-number*
- States used in multiple blocks by using the format *block-name: state-name*

To configure an individual signal line in the Code Mappings editor for a model, first you must add the signal to the mappings. See “Configure Signal Data for C Code Generation”.

Storage Class — Code definition for signal

Default (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer | dictionary
defined storage class (only for ERT models with data interface configuration)

Storage class definitions to use for generating code from the signal or state. For more information, see “Choose Storage Class for Controlling Data Representation in Generated Code”.

Path — Path to signal line or state in model

character vector

Link that you can click to highlight the signal line or block that uses the state in the model diagram.

Identifier — Name of variable

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name for the variable that represents the signal or state in the generated code.

Calibration Access — Enable or disable calibration of the signal or state

NoCalibration (default) | Calibration

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Select Calibration for the signal or state to enable the calibration, or NoCalibration to view the value of the signal or state and disable the calibration.

Display Identifier — Display name of the signal or state

character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Optional display name of the signal or state for the measurement purpose in the calibration tool, which is different from the signal or state name in the Simulink model.

Format — Display format of the signal or state value

%[length].[layout]

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Special display format to be specified for measurement in the calibration tool. This format specification overrides the display format specified in Compu Method of the signal or state.

Compu Method — Name of conversion method
character vector

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Name of the method for converting the ECU-internal value to a physical value for easy readability.

Measurement Service (only applicable for ERT models with data interface configuration) —
Name of measurement service interface
Dictionary default (default) | Not measured | dictionary defined measurement service

To specify the value of this property, click the pencil icon  at the end of the table entry to open a dialog-box that contains the property.

Use the drop-down list to specify that the signal or state should not be measured, or select a measurement service, from those defined in the dictionary, to use for measuring the signal or state. For more information, see “Configure Measurement Service Interfaces for Signals, States, and Data Stores”.

Version History

Introduced in R2018a

See Also

`coder.mapping.api.CodeMapping` | Embedded Coder Dictionary

Topics

“C Data Code Interface Configuration for Model Interface Elements”

“Choose Data Configuration Approach”

“Choose Storage Class for Controlling Data Representation in Generated Code”

“Configure Default C Code Generation for Categories of Data Elements and Functions”

“Configure Generated C Function Interface for Model Entry-Point Functions”

“Customize Generated Entry Point C Functions”

“Configure Signal Data for C Code Generation”

“Customize Generated Entry Point C Functions”

Code Replacement Viewer

Explore content of code replacement libraries

Description (Collapsed Portion)

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, <code>cos</code> or <code>RTW_OP_ADD</code>).
Implementation	Name of the implementation function, which can match or differ from Name .
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, <code>cos</code> or <code>RTW_OP_ADD</code>), 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: <code>FCN_IMPL_FUNCT</code> for function or <code>FCN_IMPL_MACRO</code> for macro.
Saturation mode	Saturation mode that the implementation function supports. One of: <code>RTW_SATURATE_ON_OVERFLOW</code> <code>RTW_WRAP_ON_OVERFLOW</code> <code>RTW_SATURATE_UNSPECIFIED</code>

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.

Field	Description
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

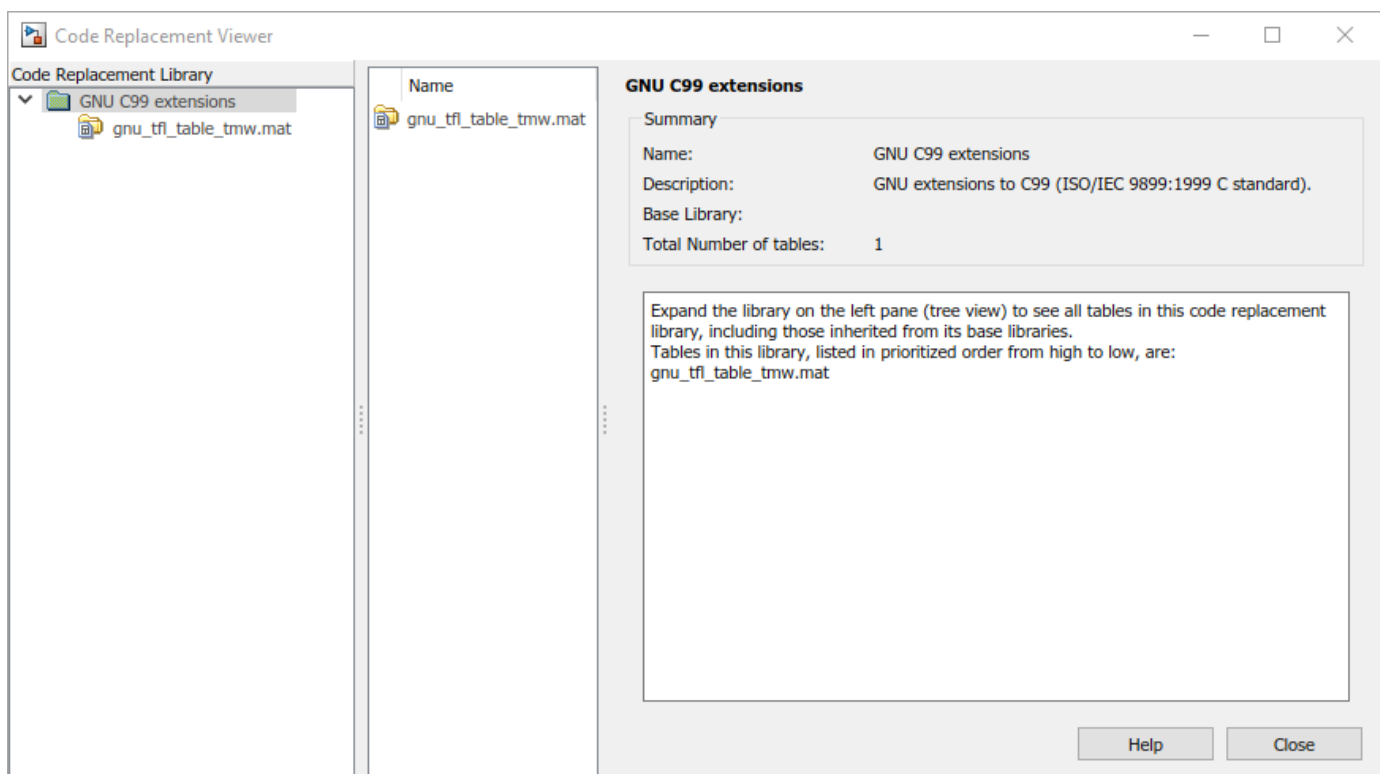
Open from the MATLAB command prompt using `crviewer`.

Examples

Display Contents of Code Replacement Library

This example opens the registered code replacement library GNU C99 extensions.

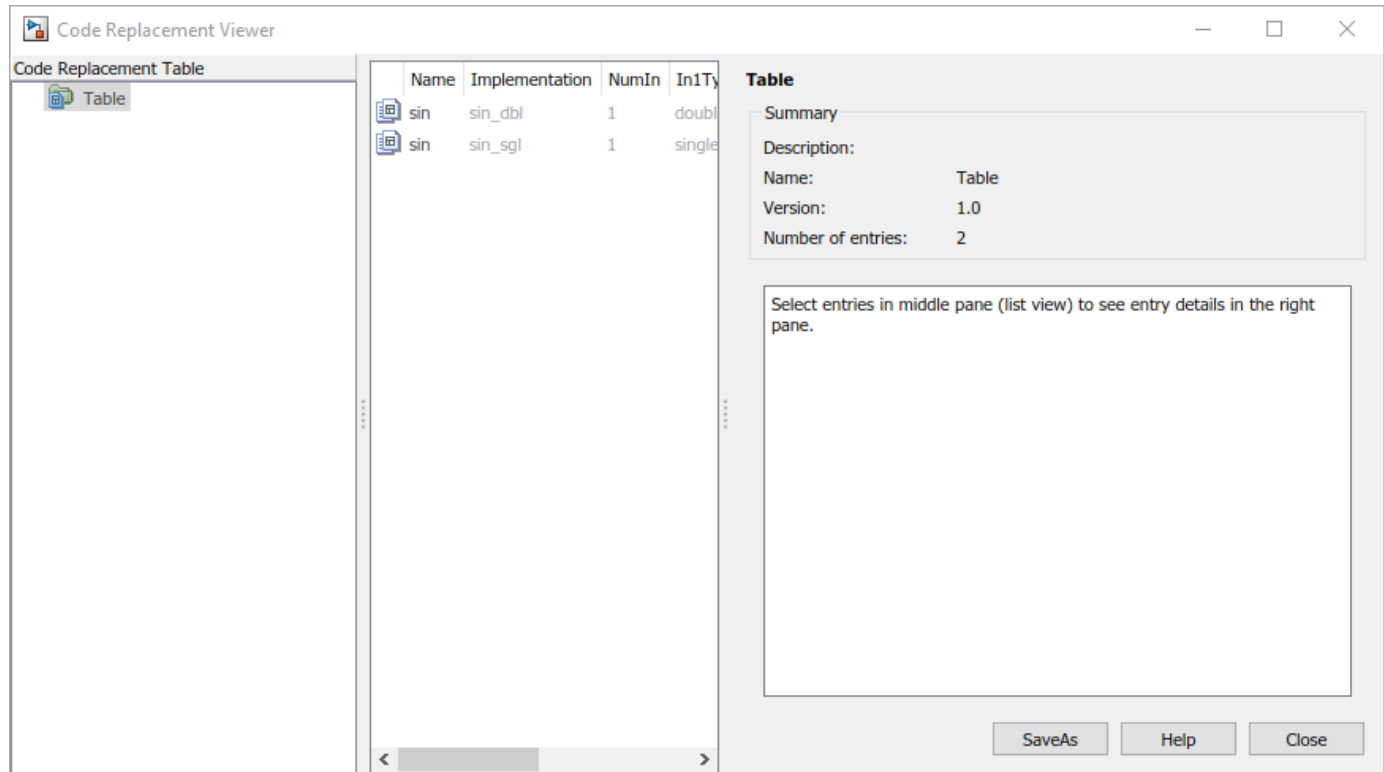
```
crviewer('GNU C99 extensions')
```



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 Library Optimizations”.

```
crviewer(crl_table_sinfcn)
```



- “Verify Code Replacement Library”

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.

Version History

Introduced in R2014b

See Also

Topics

- “Verify Code Replacement Library”
- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”

Protected Model Creator

Specify model protection options and protect model

Description

The Protected Model Creator lets you create a protected model (.slxp).

When you protect a model, you can allow the recipient of the protected model to:

- Simulate a model that includes the protected model in normal, accelerator, rapid accelerator, or external mode.
- Open a read-only web view of the model, including model contents and block parameters. Creating a read-only web view of the model requires a Simulink Report Generator license.
- Generate C and C++ code for a model that includes the protected model, if you have a Simulink Coder license.
- Generate HDL code for a model that includes the protected model, if you have an HDL Coder license.
- Generate code for the protected model through the standalone interface, if you have an Embedded Coder license and specify an ERT-based system target file for the model.

You can optionally password-protect each option. If you choose password protection for one of these options, the software protects the supporting files by using AES-256 encryption. Before you create a protected model, consider how the third party will use the model. For best practices, see “Best Practices for Creating Protected Models”.

Protected Model Creator: sldemo_mdref_counter

Description

Create a protected model (.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.

General | **Simulation Options** | **Save Options**

Supported functionality

☐ Read-only view Enter password (optional) Re-enter password (optional)

☒ **Simulation** Enter password (optional) Re-enter password (optional)

☐ Code generation Enter password (optional) Re-enter password (optional)

Content type: Binaries

☐ HDL code generation Enter password (optional) Re-enter password (optional)

Identifier tag:

Create **Cancel** **Reset** **Help**

Open the Protected Model Creator

How you open the tool depends on whether you want to protect a referenced model or the top model.

- To protect a referenced model, select a Model block that references the model you want to protect. In the Simulink Toolstrip, on the **Model Block** tab, click **Protect**.
- To protect the top model, in the Simulink Toolstrip, on the **Simulation** tab, click the **Save** button arrow. Then, select **Protected Model**.

Examples

- “Protected Model Capabilities”

- “Protect Circuit Envelope Model” (RF Blockset)

Parameters

General

Read-only view — Option to support read-only web view

off (default) | on

Specify whether to share an interactive rendition of a Simulink model that the recipient can view in a web browser with optional password protection. This read-only web view of the model includes the contents and block parameters of the model.

- **on** — Share a web view of the model that you protect. For password protection, create and verify a password with a minimum of eight characters.
- **off** — Do not share a web view of the model that you protect.

To open a web view of a protected model, perform one of these actions:

- **Simulink Editor:** On the Model block, right-click the protected-model badge icon . Then, select **Show Web view**.
- **Files panel:** Double-click the protected model file (.slxp).

Dependencies

Creating a read-only web view of the model requires a Simulink Report Generator license.

Simulation — Option to support simulation

on (default) | off

Specify whether the protected model supports simulation. Optionally, add password protection.

- **on** — A model that references the protected model can run in normal, accelerator, or rapid accelerator mode. For password protection, create and verify a password with a minimum of eight characters.
- **off** — The protected model does not support simulation.

For information about simulation modes in model hierarchies, see “Choose Simulation Modes for Model Hierarchies”.

Code generation — Option to support C and C++ code generation

off (default) | on

Specify whether the protected model supports C and C++ code generation. Optionally, add password protection.

- **on** — A model that references the protected model can run in normal, accelerator, rapid accelerator, or external mode and can support C and C++ code generation. For password protection, create and verify a password with a minimum of eight characters.

- **off** — The protected model does not support C or C++ code generation.

When you select this parameter, the software automatically selects **Simulation**.

Dependencies

To generate C and C++ code, you must have a Simulink Coder license.

Code interface — Interface for generated code

`Model reference (default) | Top model`

Specify the interface for the generated code.

- **Model reference** — A Model block that references the protected model can support C and C++ code generation and simulation in accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode. For SIL and PIL simulations, the software deploys the generated code as part of a larger application that uses the referenced model. The generated artifacts match those generated by the `slbuild` function with the "ModelReferenceCoderTarget" build specification.
- **Top model** — A Model block that references the protected model can support simulation in accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode. For SIL and PIL simulations, the software deploys the generated code as a standalone application. The generated artifacts match those generated by the `slbuild` function with the "StandaloneCoderTarget" build specification.

SIL and PIL simulations can verify the protected model code.

Dependencies

To enable this parameter:

- Select **Code generation**.
- Set the **System target file** configuration parameter to an ERT-based system target file, for example, `ert.tlc`. This option requires an Embedded Coder license.

Content type — Appearance of generated code

`Obfuscated source code (default) | Binaries | Readable source code`

Specify the appearance of the generated code.

- **Binaries** — Include only compiled binaries and headers for the generated code. The binary files and headers are visible.
- **Obfuscated source code** — Include header files found on the include path. The code in the build folder and the headers referenced by the code are visible, but obfuscated.
- **Readable source code** — Include header files found on the include path. The code in the build folder, the headers referenced by the code, and the code comments are visible and readable.

This table compares the **Content type** parameter values to the corresponding `Simulink.ModelReference.protect` function argument values.

Content type Value	OutputFormat Value	ObfuscateCode Value
Binaries	'CompiledBinaries'	true

Content type Value	OutputFormat Value	ObfuscateCode Value
Obfuscated source code	'AllReferencedHeaders'	true
Readable source code	'AllReferencedHeaders'	false

Tips

To include only the minimal header files required to build the code with the chosen build settings, use the `Simulink.ModelReference.protect` function with `OutputFormat` set to `'MinimalCode'`.

Dependencies

To enable this parameter, select **Code generation**.

HDL code generation — Option to support HDL code generation

off (default) | on

Specify whether the protected model supports HDL code generation. Optionally, add password protection.

- **on** — A model that references the protected model can run in normal, accelerator, or rapid accelerator mode and can support HDL code generation. For password protection, create and verify a password with a minimum of eight characters.
- **off** — The recipient cannot generate HDL code for the protected model.

When you select this parameter, you also select **Simulation**.

Dependencies

To generate HDL code, you must have an HDL Coder license.

Identifier tag — Custom text string for additional information

no default

Specify a custom text tag with information that is relevant to the protected model.

Being able to trace a protected model to the corresponding version of the design file can be useful for debugging. To store file provenance information in a protected model, specify a custom text tag with information such as the job number, Git commit, or continuous integration system tag associated with creating the protected model.

The tag length must not exceed 256 characters.

Simulation Options

Tunable parameters for simulation — Option to support tunable parameters

no default

Select the parameters from external data sources that you want the recipient to be able to tune during simulation.

The model protection process honors the tunability of parameters in the model workspace. Model arguments are inherently tunable. You do not need to specify the tunability of model arguments.

To compile the model and list the tunable parameters from external data sources, click **Refresh**.

Select the tunable parameters individually, or click **Select All**.

For more information about tunable parameters for simulation, see “Tune and Experiment with Block Parameter Values”.

For code generation, parameters with non-automatic storage classes are tunable. For more information, see “Storage Classes for Parameters and Signals Used in Model Blocks”.

Save Options

Destination folder — Folder to contain generated files

current working folder (default)

Specify the path of the folder to contain the generated files.

Dependencies

The model being protected must be on the MATLAB path. For more information, see “What Is the MATLAB Search Path?”

Package protected model with dependencies in a project — Option to package protected model in project

on (default) | off

Specify whether to package supporting files, including a harness model, with the protected model in a project archive. The type and number of supporting files depends on the model being protected. Examples of supporting files are a MAT file with base workspace definitions and a data dictionary pruned to relevant definitions. The supporting files are not protected.

- **on** — Create a project archive (.mlproj) that contains the protected model, its dependencies, and its harness model. The supporting files are not protected. The project archive is a single file that allows for easy sharing. To create the interactive project, open the project archive.

Set **Contents** to Protected model (.slxp) and dependencies in a project.

- **off** — Create only the protected model. If the protected model has dependencies, you must share them separately. Similarly, if you create a harness model for the protected model, you must share the harness model separately.

Set **Contents** to Protected model (.slxp) only.

Tips

Before sharing the project, check whether the project contains the required supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify the missing files. Add the missing dependencies to the project and update the harness model as needed. For more information, see “Manage Project Files”.

Dependencies

To enable this parameter, select **Simulation**.

Create harness model for protected model — Option to create harness model`on (default) | off`

Specify whether to create a harness model for the protected model. The harness model provides an isolated environment for the protected model, which is referenced by a Model block.

- `on` — Create a harness model for the protected model.
- `off` — Do not create a harness model for the protected model.

When you create a harness model for a protected model that relies on base workspace definitions, Simulink creates a MAT file that contains the required base workspace definitions.

The harness model must have access to supporting files, such as a MAT file with base workspace definitions or a data dictionary.

Dependencies

To clear this parameter, perform one of these actions:

- Clear **Package protected model with dependencies in a project**.
- Clear **Simulation**.

Name of project archive (.mlproj) — Name of project and project archive`modelName_protected (default) | valid model name`

Specify the name of the project archive that contains the generated files. The project inside the archive uses the same name.

If you do not specify a custom name, the default name for the project is the protected model name followed by `_protected`.

Dependencies

To enable this parameter, select **Package protected model with dependencies in a project**.

Tips

- To create a protected model programmatically, use the `Simulink.ModelReference.protect` function.
 - This function provides options that the Protected Model Creator does not support. For example, you can create a modifiable protected model, add callbacks to the protected model, sign the protected model with a digital certificate, and add a postprocessing function for protected model files.
 - When you create a protected model with this function, you must specify whether to generate the protected model report. The function does not generate this report by default.
- The Protected Model Creator caches your settings for a model during a MATLAB session. If you close and reopen the Protected Model Creator, the cached settings persist. To restore the default settings, click **Reset**. Passwords and tunable parameter selections are not cached.

Version History

Introduced in R2012b

R2024b: Specify custom text tag with additional information

The new **Identifier tag** parameter lets you specify custom text with information that is relevant to the protected model.

Being able to trace a protected model to the corresponding version of the design file can be useful for debugging. To store file provenance information in a protected model, specify a custom text tag with information such as the job number, Git commit, or continuous integration system tag associated with creating the protected model.

R2024b: Model protection now provides limited build process information

To focus on the high-level steps, the model protection process now displays limited build process information by default. The model protection process temporarily overrides the values of the **Verbose accelerator builds** and **Verbose build** configuration parameters.

To display verbose build process information during model protection, use the `Simulink.ModelReference.protect` function and set `Verbose` to `true`.

R2024b: Protected model harness uses default instance parameter values

Behavior changed in R2024b

When you create a harness model for a protected model that has model arguments, the harness model uses the default instance parameter values for the Model block. Previously, the harness model inconsistently applied the instance parameter values.

R2024a: Renamed from Create Protected Model

The Create Protected Model dialog box is now called the Protected Model Creator. The available options remain the same.

Tabs now group the model protection options based on your goals:

- **General** — Specify the functionality that the protected model supports. Include optional password protection.
- **Simulation Options** — Specify the parameters that recipients of the protected model can tune during simulation.
- **Save Options** — Specify where to save the protected model, whether to save the protected model in a project with its supporting files, and whether to create a harness model.

Some options have new names.

- **Open read-only view of model** is now **Read-only view**.
- **Simulate** is now **Simulation**.
- **Use generated code** is now **Code generation**.
- **Use generated HDL code** is now **HDL code generation**.

The **Contents** menu is replaced by the **Package protected model with dependencies in a project** check box.

R2023b: Settings cached during MATLAB session

The Create Protected Model dialog box caches your settings for a model during a MATLAB session. If you close and reopen the dialog box, the cached settings persist. To restore the default settings, click **Reset**. Passwords and tunable parameter selections are not cached.

R2023b: Tunable parameters populated on demand

The Create Protected Model dialog box populates tunable parameters on demand. To compile the model and list tunable parameters, click **Refresh**.

R2023b: External mode simulation support

When you select **Use generated code**, the protected model supports external mode simulation.

R2022a: Specify tunable parameters for simulation

When you create a protected model, you can specify which parameters recipients can tune during simulation. Select the tunable parameters individually, or click **Select All**.

For code generation, parameters with non-automatic storage classes are tunable.

R2020b: Automatically package protected models with their dependencies

When you create a protected model, you can now automatically package it with its dependencies and a harness model in a project archive. When the recipient extracts the contents of the project archive and opens the harness model, they should be able to simulate the protected model without needing to define missing variables or objects. Before sharing the project archive, check whether the project contains the required supporting files, and update the harness model as needed.

In the Create Protected Model dialog box, set **Contents** to **Protected model (.slxp) and dependencies in a project**. For **Name of project archive (.mlproj)**, use the default name or specify a name. The project inside the project archive uses the same name.

For more information, see “Package and Share Protected Models”.

See Also

`Simulink.ModelReference.protect`

Topics

- “Protected Model Capabilities”
- “Protect Circuit Envelope Model” (RF Blockset)
- “Protect Models to Conceal Contents”
- “Best Practices for Creating Protected Models”
- “Reference Protected Models from Third Parties”

“Create Projects”

Generate Calibration Files

Generate ASAP2 and CDF files

Description

Use the Generate Calibration Files tool to customize and generate the calibration files for a Simulink model. You can generate an ASAP2 file, a CDF file, or both. By default, the tool generates the latest supported version of the A2L file according to ASAM MCD-2 MC standards. You can choose to generate a different version. You can customize the options for calibration files, such as the version for the A2L file and the schema type, for the CDF file. The tool remembers the last used settings, such as the version of the ASAP2 file, the option to exclude comments, the option to exclude referenced model elements, and the option to enable the ASAP2 file and CDF file generation. Simulink saves these settings in your MATLAB settings. The saved settings override the default settings.

 Generate Calibration Files (Demo) — □ ×

Generate .a2l and .cdfx files according to "ASAM MCD-2 MC (ASAP2/A2L)" and "ASAM AE CDF" standards, respectively.

Generate ASAP2 Off ☐ On ☒

File name :

Folder path : Browse...

☐ Address replacement

Symbol file : Browse...

Version :

☒ Include comments

☐ Include AUTOSAR RTE elements

☐ Include shared elements

☒ Include referenced models

Generate CDF Off ☐ On ☒

File name :

Folder path : Browse...

Schema type :

Generate Close Help

Open the Generate Calibration Files

- 1 Open the model.
- 2 Open the Embedded Coder app. For AUTOSAR models, open the AUTOSAR Component Designer app. And for Simulink Real-Time™ models, open the Simulink Coder app.
- 3 Generate code for the model.
- 4 On the **C Code** tab, select **Share > Generate Calibration Files**.

Note For AUTOSAR models, select **Autosar > Share > Generate Calibration Files**.

Examples

Generate an ASAP2 File

Configure a model to generate the latest version of the ASAP2 file with comments included.

- 1 Open the ASAP2Demo model.
- 2 Build the model.
- 3 Open the **Generate Calibration Files** tool from the **C Code > Share** tab.

Generate Calibration Files (ASAP2Demo)

Generate .a2l and .cdfx files according to "ASAM MCD-2 MC (ASAP2/A2L)" and "ASAM AE CDF" standards, respectively.

Generate ASAP2 Off ☒ On

File name :

Folder path : Browse...

☐ Address replacement

Symbol file : Browse...

Version : 1.71 ▼

☒ Include comments

☐ Include AUTOSAR RTE elements

☐ Include shared elements

☒ Include referenced models

Generate CDF Off ☐ On

File name :

Folder path : Browse...

Schema type : DTD ▼

Generate Close Help

- 4 Click the **Generate** button.

The tool generates the ASAP2Demo A2L file and saves it in the model build folder.

- "Generate ASAP2 and CDF Calibration Files"
- "Frequently Asked Questions About ASAP2 File Generation"

Version History

Introduced in R2021b

See Also

`coder.asap2.export`

Topics

“Generate ASAP2 and CDF Calibration Files”

“Frequently Asked Questions About ASAP2 File Generation”

Embedded Coder Tools

Embedded Coder Dictionary

Create code definitions to control code generation for model data and functions

More (Collapsed)

The Embedded Coder Dictionary has a pane for each type of code definition. Use the right pane to configure properties for each definition. 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 using the Code Mappings editor (see “Configure Default C Code Generation for Categories of Data Elements 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 for External Data Objects”).

Open the Embedded Coder Dictionary

- To open an Embedded Coder Dictionary, in a model window, open the Embedded Coder app by opening the **Apps** gallery and clicking **Embedded Coder**. On the **C Code** tab, select **Code Interface > Embedded Coder Dictionary**. The Embedded Coder Dictionary window displays the code interface configuration that is stored in the dictionary.
- 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 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 model RollAxisAutopilot by entering this in the MATLAB Command Window:

```
openExample("RollAxisAutopilot");
```
- 2 If the model does not open in the Embedded Coder app, open the app and click the **C Code** tab.
- 3 On the **C Code** tab, select **Code Interface > Embedded Coder Dictionary**. The Embedded Coder Dictionary window displays code generation definitions that are stored in the model file.
- 4 On the left pane, click **Storage Class**. In the **Storage Classes** section, click **Create**.
- 5 Select the new storage class that appears at the bottom of the list, **StorageClass1**. In the pane on the right, set the property values listed in this table.

Property	Value
Name	InternalStruct
Header File	internalData_\$.h
Definition File	internalData_\$.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.

- 6 Return to the model editor. To open the Interface editor, below the canvas, double-click **Code Mappings**. On the **Data Defaults** tab, expand the **Signals** section. Select the **Signals, states, and internal data** row and set **Storage Class** to InternalStruct.
- 7 In the Configuration Parameters dialog box, on the **Code Generation > Code Placement** pane, set **File packaging format** to Modular.
- 8 Generate code.
- 9 In the Simulink Editor Code view, open and inspect the file `internalData_RollAxisAutopilot.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_;
```

- 10 Open and inspect the file `internalData_RollAxisAutopilot.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 that has 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 `MultirateMultitaskingBareBoard`.
- 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, open the Embedded Coder app and open the Embedded Coder Dictionary.

5 In the Embedded Coder Dictionary, on the **Function Customization Template** tab, click **Create**.

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 In the model window, open the Code Mappings editor. On the **Function Defaults** tab, for the **Initialize/Terminate** and **Execution** rows, set **Function Customization Template** to myFunctions.

8 Generate code.

9 In the Code view, open and inspect the file MultirateMultitaskingBareBoard.c. The file defines the two execution functions, func_step0_MultirateMultitaskingBareBoard and func_step1_MultirateMultitaskingBareBoard, whose names conform to the rule that you specified in the function template.

To learn more about how to use function customization templates to customize generated C function names, see “Customize Generated Entry-Point C Function Names” and “Customize C Functions Generated from Global Simulink Functions”.

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 **Model 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** initialization. Consider creating one copy of the storage class for parameter data (**Static**) and one copy for other data (**Dynamic**).

Create Storage Class

1 Open the model RollAxisAutopilot by typing this command:

```
openExample("RollAxisAutopilot");
```

2 If the **C Code** tab is not open, open the Embedded Coder app and click the **C Code** tab.

3 Select **Code Interface > Embedded Coder Dictionary**.

4 On the left pane, click **Storage Class**. In the **Storage Classes** section, click **Create**.

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
- **Data Initialization** to Dynamic

After making your changes, verify that the pseudocode preview reflects what you expect.

- 6 Place your cursor over the row for the storage class **SigsStates** and click **Duplicate code definition**. A new storage class, **SigsStates_copy**, appears.
- 7 For the new storage class, set these properties:

- **Name** to **Params**
- **Data Initialization** to **Static**

Verify that the pseudocode preview reflects what you expect.

Apply Storage Class and Generate Code

- 1 In the model, open the Code Mappings editor. Below the model canvas, double-click **Code Mappings - Component Interface**.
- 2 On the **Data Defaults** tab, for the **Parameters > Model Parameters** row, in the **Storage Class** column, select **Params**.
- 3 For the **Signals > Signals, states, and internal data** row, set **Storage Class** to **SigsStates**.
- 4 Configure some parameter data elements in the model so that optimizations do not eliminate those elements from the generated code. On the **Modeling** tab, click **Design > Model Workspace**.
- 5 In the Model Explorer, on the center pane, select the three rows that correspond to the variables **dispGain**, **intGain**, and **rateGain** in the model workspace.
- 6 Right-click one of the rows and click **Convert to parameter object**. The Model Data Editor converts the workspace variables to **Simulink.Parameter** objects.
- 7 In the row for the parameter **dispGain**, in the **Storage Class** column, click **Configure**. The model window highlights the row for the **dispGain** parameter in the Code Mappings editor.
- 8 For each variable, in the **Storage Class** column, select **Model default: Params**, which means that they acquire the default storage class that you specified for **Model parameters**.
- 9 In the Configuration Parameters dialog box, on the **Code Generation > Code Placement** pane, set **File packaging format** to **Modular**.
- 10 Generate code.
- 11 In the Code view, open and inspect the files **RollAxisAutopilot_my_data.c** and **RollAxisAutopilot_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 for External Data Objects”). Those definitions then appear available for selection in the Code Mappings editor. In this example, you configure the Embedded Coder Dictionary in **RollAxisAutopilot** to refer to definitions stored in the built-in example package **ECoderDemos**.

- 1 Open the Embedded Coder Dictionary for **RollAxisAutopilot**. For instructions, see “Create and Verify Storage Class” on page 23-2.

- 2 In the Embedded Coder Dictionary window, on the **Memory** tab, 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 `RollAxisAutopilot`, 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**.

For an example that shows how to share code generation definitions between models by using data dictionaries, see “Share Code Interface Configuration Between Models”.

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 right pane in the Embedded Coder Dictionary window.

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.

Data Source — Location of storage class definition

text

This parameter 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?”).
- **Package name** — Defined in the Simulink package or in a custom package (see “Create Storage Classes by Using the Custom Storage Class Designer”).

Data Access — Specification to access the data

`Direct` (default) | `Function` | `Pointer`

Specification to access data associated with the model. Access the data directly (`Direct`), through customizable `get` and `set` functions (`Function`), or by using a pointer (`Pointer`). For more

information, see “Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary”.

Dependencies

- Setting this property to **Pointer**:
 - Sets the **PreserveDimensions** property to **false**. To preserve dimensions of multidimensional arrays in the generated code, set **DataAccess** to **Direct** or **Function**.
- Setting this property to **Function** or **Pointer**:
 - Sets **DataScope** to **Imported**.
 - Means that you cannot specify multi-instance properties.

In addition, setting this property to **Function** enables these properties:

- **AccessMode**
- **AllowedAccess**
- **GetFunctionName**
- **SetFunctionName**

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
\$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 <code>step</code>
\$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 <code>step</code>
\$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.

Use bit fields — Specification to represent structure members as bit fields

off (default) | on

Specification for the storage class to represent members of the structure as bit fields in the generated code.

This setting only applies to Boolean, fixed-point, and integer data types. Data types whose size exceed the size of an integer are not supported.

Storage classes with this property enabled cannot be assigned to data using the **Default Code Mappings** tab.

Dependencies

Setting **Data Access** to **Direct**, **Data Scope** to **Exported**, and **Storage Type** to **Structured** enables this property.

Data Initialization — How to initialize data

Auto (default) | Dynamic | Static | None

Specification that the generated code initialize the data.

- **Auto** — The generated code statically initializes parameter data and dynamically initializes signal and state 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.

For storage classes used for persistent data, you must set **Data Initialization** to **None**.

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.

If you select this property, you cannot set **Data Initialization** to **Dynamic**.

Dependencies

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
<code>ModelFunction</code>	In the Code Mappings editor, use for entry-point functions for initialization, execution, termination, and reset (see “Configure Default Code Generation for Functions”)
<code>UtilityFunction</code>	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.

Data Source — Location of function template definition

`text`

This parameter 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?”).

Function Naming Rule — 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
<code>\$R</code>	Name of root model
<code>\$N</code>	Base name of associated function, such as <code>step</code>
<code>\$U</code>	User token text, which you specify for a model as described in “Identifier Format Control”
<code>\$C</code>	For shared utility functions, a checksum inserted to avoid name collisions
<code>\$M</code>	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. Memory sections that are stored in a package that the dictionary references are not supported. For information about memory sections, see “Control Data and Function Placement in Memory by Inserting Pragmas”.

Header File — Name of header file that declares function

`text`

Name of the header file that declares the function. You can use these tokens in the header filename.

Token	Description
<code>\$R</code>	Name of root model
<code>\$N</code>	Base name of associated function, such as <code>step</code>
<code>\$U</code>	User token text, which you specify for a model as described in “Identifier Format Control”

For more information about using a custom header file, see “Control File Packaging of Generated Entry-Point Functions and Simulink Functions”.

Definition File — Name of source file that defines function

`text`

Name of the source file that defines the function. You can use these tokens in the definition filename.

Token	Description
<code>\$R</code>	Name of root model

Token	Description
\$N	Base name of associated function, such as <code>step</code>
\$U	User token text, which you specify for a model as described in “Identifier Format Control”

For more information about using a custom definition file, see “Control File Packaging of Generated Entry-Point Functions and Simulink Functions”.

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 following memory sections from the Simulink package.

Memory Section	Description
MemConst	Apply the storage type qualifier <code>const</code> to the data.
MemVolatile	Apply the storage type qualifier <code>volatile</code> to the data.
MemConstVolatile	Apply the storage type qualifiers <code>const</code> and <code>volatile</code> to the data.

Note The listed memory sections concern the `const` and `volatile` qualifiers. These qualifiers do not apply to functions, or to **Data Defaults** categories in the Code Mappings Editor that are mapped to the `Default` storage class.

Description — Purpose and functionality of memory section

text

Custom text that you can use to describe the purpose and functionality of the memory section.

Data Source — Location of memory section definition

text

This parameter 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?”).
- Package name — Defined in the Simulink package or in a custom package (see “Create Code Definitions for External Data Objects”).

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.

You can use the token `$R` to represent the name of the model that uses 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.

You can use the token `$R` to represent the name of the model that uses 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`.

Receiver Service Interfaces

Name — Name of receiver service

text

Name of the receiver service. Embedded Coder provides the example services listed in this table.

Receiver Service	Description
ReceiverExample1	Favor memory optimization over data freshness. The service reads data from another function before the function starts executing.
ReceiverExample2	Favor data freshness over memory optimization. The service reads data from another function immediately during execution.
ReceiverExample3	Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design. The platform service reads data from generated function code directly by using memory that platform services manage for the target execution environment.

Data Communication Method — How receiver data transmission is handled

`Outside Execution (default)` | `During Execution` | `Direct Access`

How receiver data transmission is handled between the target platform service and the run-time environment, specified as one of these options:

- **Outside Execution** — Read data from another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- **During Execution** — Read data from another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.
- **Direct Access** — Read data from generated function code directly by using memory that platform services manage. Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design.

Function Naming Rule — Names of generated receiver functions

text

Names of the receiver 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
\$G	Service name
\$N	Element name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in “Identifier Format Control”
\$M	Name-mangling text inserted to avoid name collisions

Dependencies

To specify a function naming rule, you must set **Data Communication Method** to **Outside Execution** or **During Execution**.

Storage Class — Storage class definition

text

Storage class definition, specified as one of the definitions in the **Storage Class** section of the dictionary.

Dependencies

To specify a storage class receiver services, you must set **Data Communication Method** to **Direct Access**.

Sender Service Interfaces

Name — Name of sender service

text

Name of the sender service. Embedded Coder provides the example services listed in this table.

Sender Service	Description
SenderExample1	Favor data freshness over memory optimization. The platform service reads data from another function immediately during execution.

Sender Service	Description
SenderExample2	Favor memory optimization over data freshness. The service sends data to another function after the function executes.
SenderExample3	Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design. The platform service reads data from generated function code directly by using memory that platform services manage for the target execution environment.

Data Communication Method — How sender data transmission is handled

Outside Execution (default) | During Execution | Direct Access

How sender data transmission is handled between the task and the run-time environment, specified as one of these options:

- **Outside Execution** — Send data to another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- **During Execution** — Send data to another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.
- **Direct Access** — Send data to generated function code directly by using memory that platform services manage. Use this option to maximize performance of component code or if mutual exclusion is inherent in component model design.

Function Naming Rule for Value — Names of generated sender functions

text

Names of the sender 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
\$G	Service name
\$N	Element name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in “Identifier Format Control”
\$M	Name-mangling text inserted to avoid name collisions

Dependencies

To specify a function naming rule, you must set **Data Communication Method** to **Outside Execution** or **During Execution**.

Function Naming Rule for Value by Reference — Names of generated sender functions that use reference

text

Names of the sender functions that send a value by reference 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
\$G	Service name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in “Identifier Format Control”

Dependencies

To specify a function naming rule, the definition must be a data sender service and you must set **Data Communication Method** to `Outside Execution` or `During Execution`.

Storage Class — Storage class definition

text

Storage class definition, specified as one of the definitions in the **Storage Class** section of the dictionary.

Dependencies

To specify a storage class receiver services, you must set **Data Communication Method** to `Direct Access`.

Data Transfer Service Interfaces

Name — Name of data transfer service

text

Name of the data transfer service. Embedded Coder provides the example services listed in this table.

Data Transfer Service	Description
DataTransferExample1	Favor data freshness over memory optimization. The platform service reads data from another function immediately during execution.
DataTransferExample2	Favor memory optimization over data freshness. The service receives and sends data from another function outside of function execution.

Data Communication Method — How data transmission is handled

`Outside Execution (default)` | `During Execution`

How data transmission is handled between the task and the run-time environment, specified as one of these options:

- **Outside Execution** — Communicate data to another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- **During Execution** — Communicate data to another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.

Receiver Function Naming Rule — Names of generated receiver functions

text

Names of the receiver 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	Element name
\$X	Current callable function
\$M	Name-mangling text inserted to avoid name collisions

Sender Function Naming Rule — Names of generated sender functions

text

Names of the sender 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	Element name
\$X	Current callable function
\$M	Name-mangling text inserted to avoid name collisions

Timer Services**Name** — Name of timer service

text

Name of the timer service. Embedded Coder provides the example service listed in this table.

Timer Service	Description
TimerServiceExample1	The timer service executes outside of function execution.

Data Communication Method — How data transmission is handled

Outside Execution (default) | During Execution

How data transmission is handled between the task and the run-time environment, specified as one of these options:

- **Outside Execution** — Communicate data to another function before the function starts executing. Use this option to favor memory optimization over data freshness.
- **During Execution** — Communicate data to another function immediately during execution. The generated function code that calls the service must use a local buffer to maintain value coherence during execution. Use this option to favor data freshness over memory optimization.

Function Clock Tick Function Naming Rule — Clock tick function naming rule

text

Names of the clock tick 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
\$G	Service name
\$X	Current callable function
\$U	User token text, which you specify for a model as described in “Identifier Format Control”

Dependencies

- If you create multiple timer services, use the \$G token.
- If you set **Data Communication Method** to `Outside Execution`, use the \$X token.

Parameter Tuning Interfaces

Name — Name of parameter tuning service

`Not tunable` (default) | `text`

The name of the tuning service for parameters that need to be tuned. If the parameters do not need to be tuned, **Name** has the default value, `Not tunable`.

Storage Class — Name of storage class definition

`text`

Name of storage class definition to use with the tuning service, specified as one of the compatible definitions in the **Storage Class** section of the dictionary.

Note The **Storage Class** is not applicable for the **Not tunable** entry.

Parameter Argument Tuning Interfaces

Name — Name of parameter argument tuning service

`Not tunable` (default) | `text`

The name of the tuning service for parameter arguments that need to be tuned. If the parameter arguments do not need to be tuned, **Name** has the default value, `Not tunable`.

Storage Class — Name of storage class definition

`text`

Name of storage class definition to use with the tuning service, specified as one of the compatible definitions in the **Storage Class** section of the dictionary.

Note The **Storage Class** is not applicable for the **Not tunable** entry.

Measurement Service Interfaces

Name — Name of measurement service

`Not measured` (default) | `text`

The name of the measurement service for signals, states, and internal data elements that need to be measured. If the signals, states, and internal data elements do not need to be measured, **Name** has the default value, `Not measured`.

Persistent Data — Option to use persistent data

off (default) | on

Option to store internal state and data store data in nonvolatile memory in the target environment.

When specifying **Persistent Data**, only storage classes with **Data Initialization** set to **None** are supported.

Storage Class — Name of storage class definition

text

Name of storage class definition to use with the measurement service, specified as one of the compatible definitions in the **Storage Class** section of the dictionary.

Note The **Storage Class** for the **Not measured** entry cannot be configured. Model elements that are specified as **Not measured** and cannot be optimized away will be generated according to the **Storage Class** specified for the **Internal Data** section in the dictionary.

Dependencies

Only storage classes with **Data Initialization** set to **None** are available for service interfaces that specify persistent data.

Limitations

- 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 23-5). Use a memory section defined in the Embedded Coder Dictionary.
- A storage class that you create in an Embedded Coder Dictionary that has **Function** data access is not supported for the storage class of a data store, parameter argument tuning interface, or measurement interface.
- 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”.

Version History

Introduced in R2018a

R2025a: Measurement service interface for persistent data support for target platform nonvolatile memory

Specify a measurement service interface for persistent data to support state and data store data that is stored in nonvolatile memory in the target environment.

To specify persistent data, use the service interface **PermanentRAM** in the **Measurement Service Interfaces** section of the coder dictionary. Alternatively, you can define a new measurement service interface for persistent data in the **Measurement Service Interfaces** section by creating a new service

interface, selecting **Persistent Data**, and choosing a storage class that has **Data Initialization** set to **None**. A new storage class, **PersistentMemory**, is provided in the **Storage Class** section of the coder dictionary.

R2024a: Not measured and Not tunable as dictionary default

You can now specify **Not measured** as the default value of the **Measurement Service Interfaces** section in the coder dictionary, and you can specify **Not tunable** as the default value of the **Parameter Tuning** and **Parameter Argument Tuning** sections in the dictionary. The **Not measured** and **Not tunable** entries are included in the dictionary, and are preselected as default values in newly created dictionaries. You can select other entries as dictionary default for these sections, but you cannot delete these entries from the dictionary.

R2024a: Changes in service interfaces dictionary sections

- **Constants** and **Internal Data** are new dictionary sections.
- The dictionary sections **Subcomponent Initialize and Terminate Functions** and **Subcomponent Periodic and Aperiodic Functions** replace the section **Subcomponent Functions**.

R2022b: Create service interface configuration definition with service interfaces

When you create a shared coder dictionary in an SLDD file, you create either a data interface configuration or a service interface configuration definition in the dictionary:

- **Service interface configuration** — Create service interface definitions that describe the interfaces of the services that the platform middleware provides. The generated code uses the service interface definitions to generate calls to the services. You can define these new service interfaces:
 - Initialize and terminate functions
 - Periodic and aperiodic functions including model step functions
 - Receiver service interfaces
 - Sender service interfaces
 - Data transfer service interfaces
 - Timer service interfaces
 - Parameter tuning service interfaces
 - Parameter argument tuning service interfaces
 - Measurement service interfaces
 - Subcomponent functions
 - Shared utility functions

In the service interface configuration, you can still define storage classes and memory sections and apply them to service interface definitions.

- **Data interface configuration** — The generated code uses the data and function interfaces that you define.

Existing Embedded Coder Dictionaries contain a data interface configuration with the code definitions.

R2022b: Dictionary refreshes when you load model from earlier release*Behavior changed in R2022b*

Package-based code definitions have changed. If your Embedded Coder Dictionary refers to code definitions that you store in a package, the dictionary refreshes when you load a model from an earlier release. To prevent the dictionary refresh, resave the model, or the Simulink data dictionary that contains the Embedded Coder Dictionary, in the current release.

R2021b: Pointer data access for storage classes

When you create an unstructured storage class, you can configure the storage class to use pointer data access. Set **Data Scope** to **Imported** and set **Data Access** to **Pointer**.

R2021a: Automatic data initialization for storage classes

For new storage classes, the default **Data Initialization** setting is now **Auto**. With this setting, you can apply the same storage class to signals, states, and parameters. The generated code statically initializes parameter data and dynamically initializes signal and state data.

Previously, the options for the **Data Initialization** setting were limited to **Dynamic**, **Static**, and **None**.

R2020a: \$R token for memory sections

You can now use the \$R token in the **Pre Statement** and **Post Statement** for a memory section. \$R expands to the model name. Previously, you could use only the \$N token for memory sections. \$N expands to the name of the data element or function.

R2019a: Preserve array dimensions in storage classes

To preserve the dimensions of array data that uses a storage class, select the **Preserve array dimensions** property for the storage class.

R2019a: get and set data access for storage classes

You can now define a storage class for root-level inports, root-level outports, and local parameters so that they can be accessed by customizable **get** and **set** functions. Set the new **Data Access** property to **Function**. You can also configure these new properties:

- **Access Mode** — Whether the access functions return data by value or by pointer
- **Allowed Access** — Whether to allow read and write access, read-only access, or write-only access to the data
- **Name of Getter** — Naming rules for the **get** functions
- **Name of Setter** — Naming rules for the **set** functions

R2019a: Storage classes with different settings for single-instance and multi-instance data

You can now create storage classes that use different settings for single-instance and multi-instance data. Select the new property **Use different property settings for single-instance and multi-**

instance data. For single-instance data, you can specify the storage type and structure properties. You can separately specify the structure properties for multi-instance data.

When you apply the storage class to a data item, the Embedded Coder Dictionary implements either the single-instance settings or the multi-instance settings depending on the type of data and the context of the model within the model reference hierarchy.

R2019a: Load multiple packages in one dictionary

You can now load and refer to code definitions from multiple packages in one dictionary.

R2019a: Use code definitions from local and shared Embedded Coder Dictionaries

When a shared Embedded Coder Dictionary is attached to the model, the model can use the code definitions from both the shared dictionary and the local Embedded Coder Dictionary of the model. The local Embedded Coder Dictionary of the model displays the shared definitions as read-only.

R2018b: Restrict mapping of storage classes to parameters or signals

Specify whether users can map a storage class to parameters, signals, or parameters and signals. To allow users to map a storage class to each category, select one or both of the new properties

Parameters and **Signals**.

R2018b: Select default definitions in shared Embedded Coder Definitions

For a shared Embedded Coder Dictionary, which is saved in an SLDD file, you can select the default code definition for each category of data or functions.

R2018b: Pseudocode preview for code definitions

When you create a code definition in the Embedded Coder Dictionary, you can view a pseudocode preview of the code that will generate for the function or data item that uses the code definition.

See Also

Code Mappings Editor - C | `coder.Dictionary`

Topics

"Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture"

"Deploy Code Generation Definitions"

"Generate Code to Conform to Software Architecture by Sharing and Copying Default Settings Between Models"

"Flexible Storage Class for Different Model Hierarchy Contexts"

"Customize Generated Entry Point C Functions"

"Customize C Functions Generated from Simulink Functions"

"Customize Generated Entry Point C Functions"

"Customize C Functions Generated from Simulink Functions"

Code Mappings - C++ Editor

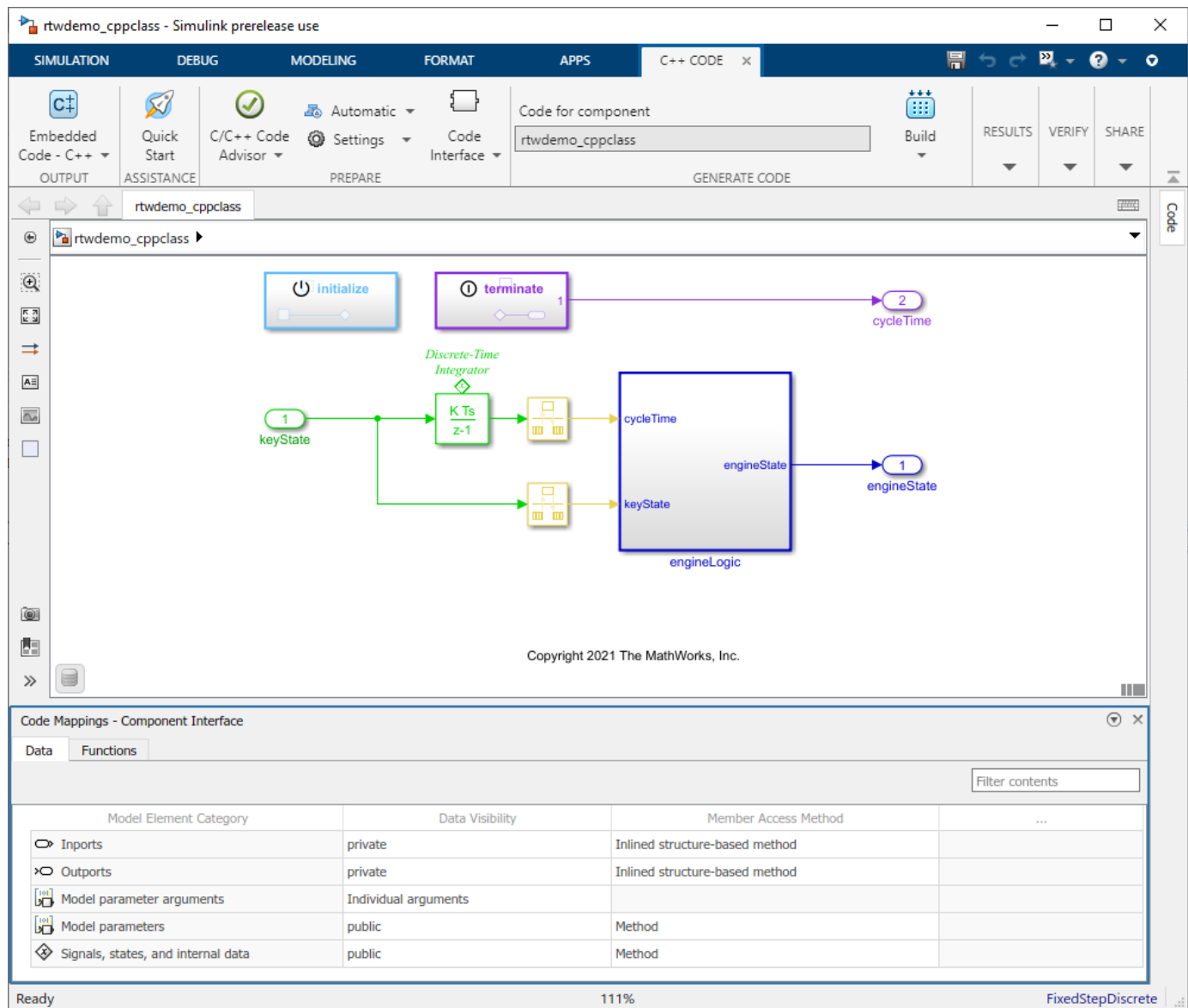
Configure how model elements and functions appear in generated C++ code

Description

The Code Mappings editor is a graphical interface used to configure how Simulink model data elements and functions appear in generated C++ code.

To configure Simulink model data elements and functions for C++ code generation, use the tables in the Code Mappings editor:

- **Data**
- **Functions**



Open the Code Mappings - C++ Editor

Open the Embedded Coder app. Verify the **Output** is set to **Embedded Code - C++**. On the **C++ Code** tab, click the **Code Interface** menu and select **Code Mappings**.

Examples

Configure Data Elements

You can use the Code Mappings editor to customize the data visibility and method access of model elements in the generated C++ class interface. This example uses the model `RollAxisAutopilot` to show how to configure the data visibility and method access for the inports in this model.

Set Up the Environment

- 1 Open model RollAxisAutopilot.

```
openExample('RollAxisAutopilot')
```
- 2 Open the Embedded Coder app. In the Apps gallery, click **Embedded Coder**.
- 3 Set the language to C++. On the **C++ Code** tab, click **Output** and select **Embedded Code - C++**.
- 4 Open the Code Mappings editor. From the tab, click **Code Interface** and select **Code Mappings**.

Customize the Data Visibility and Method Access of Data Elements

- 1 In the Code Mappings editor, click the **Data** tab.
- 2 Customize the data visibility. For the **Inports** category, in the **Data Visibility** column, select **public**.
- 3 Customize the method access. For the **Inports** category, in the **Member Access Method** column, select **method**.

Generate and Verify Code

- 1 Generate code.
- 2 Verify the generated C++ code for the inports. In the **Code** view, open RollAxisAutopilot.cpp file and search for the inports in the model.

Configure Functions

You can use the Code Mappings editor to customize the names of entry-point functions and the names and arguments of base-rate periodic or Simulink Functions. This examples uses the model RollAxisAutopilot to show how to configure the name and arguments of a base-rate periodic function.

Set Up the Environment

- 1 Open model RollAxisAutopilot.

```
openExample('RollAxisAutopilot')
```
- 2 Open the Embedded Coder app. In the Apps gallery, click **Embedded Coder**.
- 3 Set the language to C++. On the **C++ Code** tab, click **Output** and select **Embedded Code - C++**.
- 4 Open the Code Mappings editor. From the tab, click **Code Interface** and select **Code Mappings**.

Customize Function Name and Arguments

- 1 In the Code Mappings editor, click the **Functions** tab.
- 2 Customize the name of the periodic function. In the **Function Name** column, enter the name `roll_run`.
- 3 Customize the arguments of the periodic function. In the **Function Preview** column, click the prototype hyperlink. A configuration dialog box opens.
- 4 In the dialog box, select **Configure arguments for Step function prototype** and click **Get Default**. A table that displays the arguments opens.
- 5 Customize the arguments:
 - 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 Qualifier** drop-down list, select **Pointer**. In the **C++ Identifier Name** field change the name to `HDG_Mode_Ptr`
- 6 Click **Apply**. Visually verify that the function prototype reflects the changes. Click **OK** to exit.

Generate and Verify Code

- 1 Generate code.
- 2 Verify the updated method names and arguments in the generated C++ code. In the **Code** view, in the open `RollAxisAutopilot.cpp` file, search for the base-rate periodic function, `roll_run`.

Parameters

Data

Model Element Category — Category of model data elements

character vector

Each category describes a type of Simulink model data element. The data visibility and method access set for a category applies to the data elements in that category for the model.

Model Element Category	Description
Inports	Root-level data input ports of a model, such as Inport and In Bus Element blocks.
Outports	Root-level data output ports of a model, such as Outport and Out Bus Element blocks.
Model parameter arguments	Workspace variables that appear as instance (nonstatic) class data members.
Model parameters	Workspace variables that are shared across instances of the model class and are generated as static class data members.
Signals, states, and internal data	Data elements that are internal to the model, such as block output signals, discrete block states, data stores, and zero-crossing signals.

Data Visibility — Determines the access specifier for the generated class members

character vector

The data visibility determines if data elements appear in generated code as public, protected, or private. For model parameter arguments, this parameter can also specify that block parameters in referenced models are generated outside the class and passed as individual arguments.

Member Access Method — Determines the generated get and set methods for class members

character vector

The access determines how the generated code provides access to the class member data.

Functions

Source — Functions in a model

character vector

The functions in a model that generate entry-point methods in a C++ class interface. The functions include:

- Initialize Functions
- Terminate Functions
- Periodic Functions
- Partition Functions
- Exported Functions
- Reset Functions
- Simulink Functions

Method Name — Name of generated class method
character vector

Name for the generated class method.

Method Preview — Preview of method prototype
character vector

Preview of the entry-point method prototype. To customize the prototype, click the preview hyperlink and configure the method in the opened dialog box.

Version History

Introduced in R2021a

Topics

“Interactively Configure C++ Interface”

“Programmatically Configure C++ Interface”

“Configure Parameters for C++ Interface Code Generation”

Code Replacement Tool

Create, modify, and validate content of code replacement libraries

Description (Collapsed Portion)

The tool display consists of three panes that show table and table entry information:

- Left pane lists code replacement tables or libraries.
- Middle pane displays the available content of the table or library you select in the left pane. For example, if you select a table in the left pane, the table entries will be listed in the middle pane.
- Right pane displays information about the item selected in the left or middle pane. 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 you select a table entry, the right pane shows mapping and build information for that entry.

Open the Code Replacement Tool

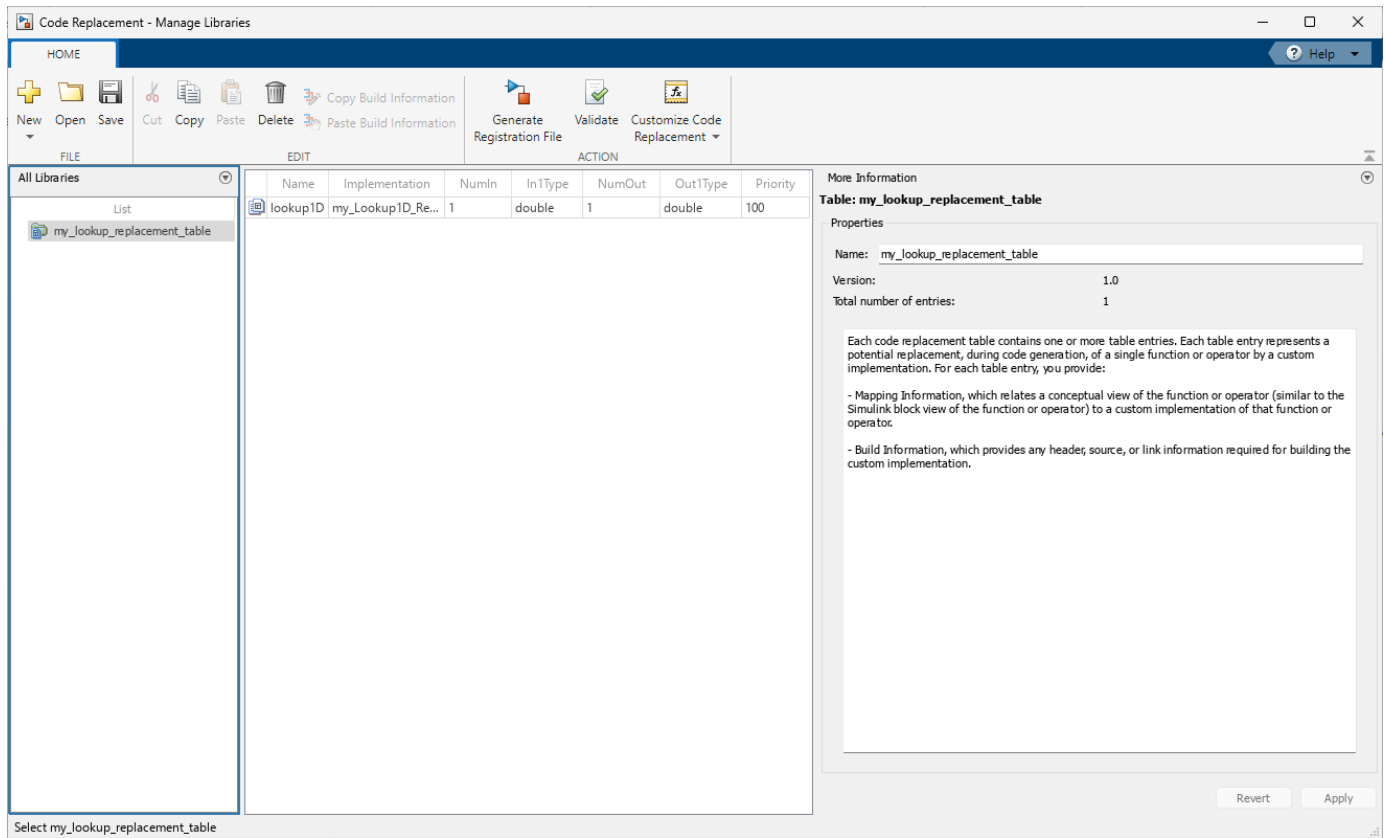
At the command prompt, type `crtool`. To view an existing code replacement table, specify that table as the argument to `crtool`. For example, if your code replacement table is called `my_code_replacement_table`, call `crtool("my_code_replacement_table.m")`.

Examples

Open an Existing Table in the Tool

This example shows how to open a code replacement table, `my_lookup_replacement_table`, in the Code Replacement Tool. This table is used in “Generate Code by Using Lookup Table Code Replacement Library”.

```
openExample("my_lookup_replacement_table.m")
crtool("my_lookup_replacement_table")
```

Code Replacement Tool Keyboard Shortcuts

You can use these keyboard shortcuts to interact with the Code Replacement Tool.

Task	Shortcut
Create table	Ctrl+T
Open table	Ctrl+O
Save selected table	Ctrl+S
Delete	Del
Copy	Ctrl+C
Cut	Ctrl+X
Paste	Ctrl+V
Copy build information	Ctrl+Shift+C
Paste build information	Ctrl+Shift+V
Create math operation entry	Ctrl+1
Create function entry	Ctrl+2
Create BLAS operation entry	Ctrl+3
Create CBLAS operation entry	Ctrl+4

Task	Shortcut
Create net slope fixed-point operation entry	Ctrl+5
Create semaphore entry	Ctrl+6
Create custom entry class	Ctrl+M
Import custom entry class	Ctrl+I
Validate selected table or entry	Ctrl+J
Generate registration file	Ctrl+G
Open help	Ctrl+H

Note When you use the keyboard shortcuts on a Mac press **command** (⌘) instead of **Ctrl**.

- “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
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.

InnType — Data type of conceptual input argument
character vector

Data type of a conceptual input argument.

NumOut — Number of output arguments (read-only)
scalar integer

Number of output arguments.

OutnType — 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. The name can be 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, sincos, or atan2, to use the approximation method CORDIC or Lookup.
- 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

[2 2] (default)

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.

Version History

Introduced in R2014b

Topics

“What Is Code Replacement?”

“What Is Code Replacement Customization?”

“Quick Start Code Replacement Library Development - Simulink”

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 Symbolic Dimensions for S-Function Blocks”

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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)]");
```

Version History

Introduced in R2016a

ssRegisterSymbolicDims

Create SymbDimsId from array of SymDimsIds

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 SymDimsIds

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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");
```

Version History

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);
```

Version History

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);
```

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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 SymbDimsIds symbDimsId and symbolId. It sets the result equal to a new SymbDimsId called outputDimsId.

```
const SymbDimsId outputDimsId =  
    ssRegisterSymbolicDimsMultiply(S, symbDimsId, symbolId);
```

Version History

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);
```

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

Introduced in R2016a

ssGetSymbolicDim

Get SymbDimsId from array of SymbDimsIds

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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);
```

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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);
```

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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 Symbolic Dimensions for S-Function Blocks”.

Version History

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.

Version History

Introduced in R2016a

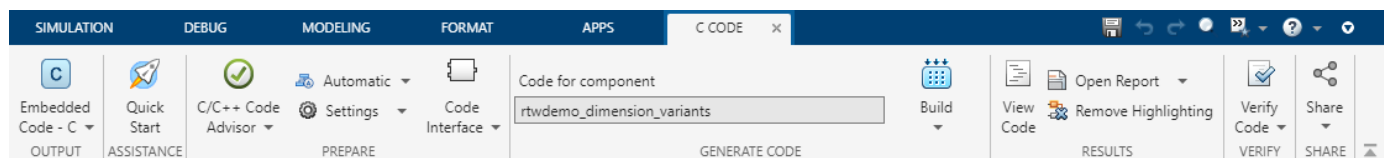
Apps

Embedded Coder

Generate readable, compact, and fast C and C++ code for embedded processors used in mass production

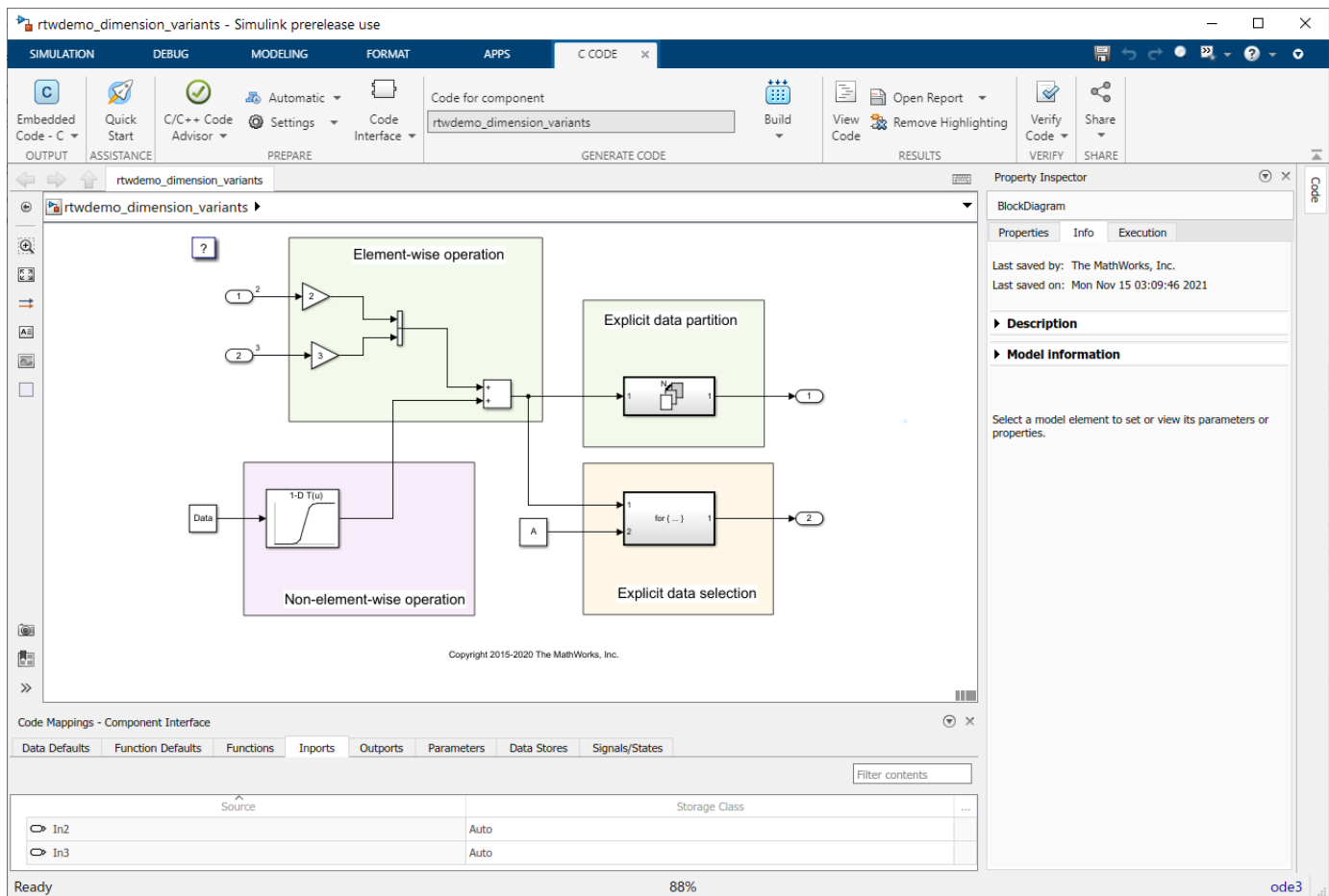
Description

Use the **Embedded Coder** app to generate C or C++ code from a model that represents a discrete-time system. The app extends the **Simulink Coder** app capabilities with advanced optimizations for precise control of the generated functions, files, and data. When you open the app, a **C Code** tab is added to the toolstrip. The **C Code** tab represents groups of tasks in the Embedded Coder workflow.



Use the app to perform these tasks:

- If you are new to Embedded Coder, use the Embedded Coder Quick Start to prepare your model for code generation. The Embedded Coder Quick Start chooses fundamental code generation settings based on your goals and application. Open the Embedded Coder Quick Start by clicking **Quick Start**.
- Set code generation objectives and prepare your model for code generation by clicking the **C/C++ Code Advisor**.
- For a model hierarchy, specify the deployment type of each model by using the **Deployment Type** drop-down. If you have not configured deployment types, the drop-down says **Automatic** because Embedded Coder determines the deployment types automatically.
- To set model configuration parameters, select **Settings > C/C++ Code generation settings** or **Settings > Hardware Implementation**.
- Opening the **Embedded Coder** app opens the Code perspective. The Code perspective contains an integrated help pane, the Code Mappings editor, and the Property Inspector or Code view. Use the Code Mappings editor to configure data elements and entry-point functions in a model. Select **Code Interface > Individual Element Code Mappings**.
- To create custom code definitions, open the Embedded Coder Dictionary by selecting **Code Interface > Embedded Coder Dictionary**.
- Generate code only by selecting **Build > Generate Code**. Build the model and generate code by selecting **Build > Build**.
- To view the generated code alongside your model, use the Code view. You can trace between model elements and the code by clicking hyperlinked lines of code in the Code view. Open the latest code generation report by selecting **Open Report**.
- Verify the equivalence of the simulation and the code execution results by opening the **SIL/PIL** app. Select **Verify Code > SIL/PIL Manager**.
- Create a protected model for simulation and code generation to share with a third party by selecting **Share > Generate Protected Model**.
- Package the code and artifacts by selecting **Share > Generate Code and Package**.



Open the Embedded Coder App

In the **Apps** gallery, under **Code generation**, click **Embedded Coder**. The **C Code** tab opens.

Examples

- “Generate Code by Using the Quick Start Tool”
- “Generate C Code from Simulink Models”
- Code Mappings Editor — C
- Embedded Coder Dictionary
- “Map Embedded System Architecture to Simulink Modeling Environment”

Tips

- If you are working with a model hierarchy, the **Embedded Coder** app automatically determines the deployment type of each model based on the position in the hierarchy. The top model is the component and referenced models are subcomponents. On the **C Code** tab, the functionalities apply to the component model. To configure the deployment types of the models in the hierarchy:

- Specify simulation only models, such as test harnesses, by clicking the down arrow in the **Output** section of the toolbar and selecting **Simulation Only**. When you view a simulation only model in the **Embedded Coder** app, the toolbar hides the code generation options.
- Specify component and subcomponent models by using the **Deployment Type** drop-down.
- To configure and view code for a referenced model, navigate to the model in the hierarchy and use the Code Mappings editor, Model Data Editor, Property Inspector, and Code view. These views apply to the active model, which can be the top model or a referenced model.

Version History

Introduced in R2019b

See Also

Functions

`coder.Dictionary`

Topics

“Generate Code by Using the Quick Start Tool”

“Generate C Code from Simulink Models”

Code Mappings Editor — C

Embedded Coder Dictionary

“Map Embedded System Architecture to Simulink Modeling Environment”

Code Profile Analyzer

Analyze execution-time and stack usage profiles for generated code

Description

Use the Code Profile Analyzer app to analyze execution-time and stack usage profiles produced by software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution of generated code:

- View execution times of tasks and functions, visualize the function-call stack at a specific simulation time, and compare execution times from different runs.
- View stack usage for tasks and functions and compare values from different runs.

If you are using Embedded Coder, you can also:

- Analyze execution-time profiles produced by XCP-based external mode simulations.
- Configure code profiling for a model hierarchy.
- Identify and analyze performance hotspots in the generated code. This workflow uses the `coder.profile.test.runTests` function.
- Identify and analyze critical paths in the generated code. This workflow uses the `coder.profile.test.analyzePath` function.

The screenshot shows the Code Profile Analyzer app interface. The top bar has tabs for GENERAL, TIME PROFILING, and a help icon. Below the top bar, there are settings for Time unit (Nanoseconds (ns)) and Timer Ticks per Second (MHz) (3595). The main area is divided into three sections: SETTINGS, ANALYSIS, and RESULTS. The ANALYSIS section is active, showing a table of Task Execution Times. The table has columns for Section Name, Minimum Execution Time, Average Execution Time, Maximum Execution Time, Average Self Time, Maximum Self Time, Calls, Standard Deviation, and Variance. The table shows data for 'initialize' and 'step [0.1 0]' tasks. Below the table, there is a section for CPU Utilization with a table showing Average CPU Utilization and Maximum CPU Utilization for the same tasks. The RESULTS section on the right shows a Summary of the profiled component (SILTopModel), workflow (SIL/PIL), environment (Simulink), save options (All data), time created (06-Jan-2025 16:33:26), and instrumentation over (Not filtered).

Section Name	Minimum Execution Time	Average Execution Time	Maximum Execution Time	Average Self Time	Maximum Self Time	Calls	Standard Deviation	Variance
initialize	290.4033	290.4033	290.4033	260.3616	260.3616	1	0	0
step [0.1 0]	40.3338	113.1495	470.6537	47.3099	80.1113	101	73.1630	5.3528e+03

Task	Average CPU Utilization	Maximum CPU Utilization
step [0.1 0]	0.0001131%	0.0004707%
Overall CPU Utilization	0.0001131%	0.0004707%

Summary

Profiled component: SILTopModel

Workflow: SIL/PIL

Environment: Simulink

Save options: All data

Time created: 06-Jan-2025 16:33:26

Instrumentation over... Not filtered

Open the Code Profile Analyzer App

If you are generating code from a Simulink model, click the **SIL/PIL** tab. In the **Results** gallery, under **Execution Profiling Results** or **Stack Profiling Results**, click the **Code Profile Analyzer** button.

If you are generating code from MATLAB code, terminate the SIL/PIL execution in the Command Window. The code generator produces a message in the Command Window that contains a link to the execution or stack profiling report. Click that link.

Alternatively, for both Simulink and MATLAB workflows, open the Code Profile Analyzer from the Command Window after the SIL/PIL execution has been terminated by entering `coder.profile.show` or `coder.profile.show(profileVariable)`.

For the Simulink workflow, *profileVariable* is a workspace variable (`coder.profile.ExecutionTime` or `coder.profile.ExecutionStack` object) that you specify through the **Workspace variable** (`CodeExecutionProfileVariable`) or **Stack workspace variable** (`CodeStackProfileVariable`) configuration parameter.

For the MATLAB workflow, obtain *profileVariable* by using the `getCoderExecutionProfile` or `getCoderStackProfile` function.

If the app is already open, you can use the **General** tab to import single simulation results by selecting the workspace variable that contains your execution-time or stack usage measurements.

You cannot use the **General** tab to import aggregated results from multiple simulations. Instead, use the command `coder.profile.show(profileAggregate)` where *profileAggregate* is a `coder.profile.ExecutionTimeSet` or `coder.profile.ExecutionStackSet` object.

Code Profile Analyzer Workflows

The type of simulation results that you import determines the tabs displayed by the app and the workflows that you can perform.

When you import results using a `coder.profile.ExecutionTime` object, the **Time Profiling** tab displays tabs as described in this table.

Tab	Purpose
Task Execution	Show execution-time metrics of tasks. For each task, inspect execution times (that is, minimum, average, and maximum execution times) and the corresponding CPU utilization, which enables you to determine whether overall the tasks can run on the target hardware without execution-time issues. See “View and Compare Code Execution Times”.
Function Execution	Show execution-time metrics of generated code functions to identify hotspots and determine functions that require rework. See “View and Compare Code Execution Times”.
Function-Call Stack	Show function-call stack at specific simulation time. If you identify that a task is taking longer than expected, you can display its function-call step for each execution instance. This feature helps you to identify performance bottlenecks. See “View and Compare Code Execution Times”.

Tab	Purpose
Comparison	Compare execution-time metrics of two simulations to identify performance regressions. See “View and Compare Code Execution Times”.
Task Validation	Perform CPU feasibility analysis to determine formally whether the task set can run without execution-time issues. Analyze the effect of task-to-CPU mapping. See “Validate Task-to-CPU Mapping”.
CPU Usage	For each task, display execution times across the available CPUs. This feature shows how tasks are allocated on the system and can help you to improve load balancing. Available only when these conditions apply: • Simulation is XCP external mode • Multi-tasking execution is enabled • Target application runs on an operating system See “Validate Task-to-CPU Mapping”.

When you import results using a `coder.profile.ExecutionTimeSet` object, the **Time Profiling** tab displays tabs as described in this table.

Tab	Purpose
Cumulative Results	Show task execution-time metrics from different tests. If you run a suite of test cases, for each task, you can identify the longest and average execution time. Then, you can identify the test case that produces the most demanding execution.
Flame Graph	Show flame graph for longest task executions. This view helps you to understand what happens during a task execution instance, enabling you to identify performance bottlenecks.
Comparison	Compare execution-time metrics of two simulations to identify performance regressions.
Task Validation	Perform CPU feasibility analysis to determine formally whether the task set can run without execution-time issues. Analyze the effect of task-to-CPU mapping. See “Validate Task-to-CPU Mapping”.

When you import results using a `coder.profile.ExecutionStack` object, the **Memory Profiling** tab displays tabs as described in this table.

Tab	Purpose
Stack Memory	Show stack usage metrics of tasks and functions. For each task, view a summary of the minimum, average, and maximum values of stack memory usage. Within a task, see how much memory is required for each executed function. See “View and Compare Stack Usage Metrics”.
Function-Call Stack	Show function-call stack at a specific simulation time. If you observe that memory allocation for a task is higher than expected, you can display its function-call stack for each execution instance. This feature helps you to identify performance bottlenecks. See “View and Compare Stack Usage Metrics”.
Comparison	Compare stack usage metrics of two simulations to identify performance regressions. See “View and Compare Stack Usage Metrics”.

When you import results using a `coder.profile.ExecutionStackSet` object, the **Memory Profiling** tab displays tabs as described in this table.

Tab	Purpose
Cumulative Results	Show task stack usage metrics from different tests. If you run a suite of test cases, for each task, you can identify the maximum value of stack usage. Then, you can identify the test case that produces the most demanding execution.
Comparison	Compare stack usage metrics of two simulations to identify performance regressions.

If your model is configured to produce Performance Monitoring Unit (PMU) metrics, then when you import results, the **Code Profiling** tab displays tabs as described in this table.

Tab	Purpose
Function Execution	Show PMU metrics of generated code functions to identify hotspots and determine functions that require rework. See “Generate Profiles Using PMU Metrics”.
Comparison	Compare PMU metrics of two simulations to identify performance regressions. See “Generate Profiles Using PMU Metrics”.

Whether you import data or not, you can also use the Code Profile Analyzer app to configure code profiling in a model hierarchy. See “Configure Execution-Time Profiling in Model Hierarchy Using Code Profile Analyzer” and “Configure Stack Usage Profiling in Model Hierarchy Using Code Profile Analyzer”.

Examples

Other Code Profile Analyzer Examples

This table gives further workflow examples.

Example	Use the app to:
"Aggregate Execution-Time Profiles to Identify Worst Execution"	Process an aggregate of execution-time profiles. For example, identify: • The longest execution for each task • Performance hotspots in generated code
"Analyze Execution Times of Critical Paths"	View results of a critical path analysis of generated code.
"Visualize Task Scheduling in XCP External Mode Simulation"	View the overrun rate (number of overrun events / number of calls) as a percentage.
"Aggregate Stack Usage Profiles to Identify Most Demanding Execution"	Process an aggregate of stack usage profiles. For example, identify the most demanding execution for each task.

- "Create Execution-Time Profile for Generated Code"
- "View and Compare Code Execution Times"
- "Validate Task-to-CPU Mapping"
- "Aggregate Execution-Time Profiles to Identify Worst Execution"
- "View and Compare Stack Usage Metrics"

Version History

Introduced in R2023a

See Also

SIL/PIL Manager

Topics

"Create Execution-Time Profile for Generated Code"

"View and Compare Code Execution Times"

"Validate Task-to-CPU Mapping"

"Aggregate Execution-Time Profiles to Identify Worst Execution"

"View and Compare Stack Usage Metrics"

Run on Custom Hardware

Run external mode simulations

Description

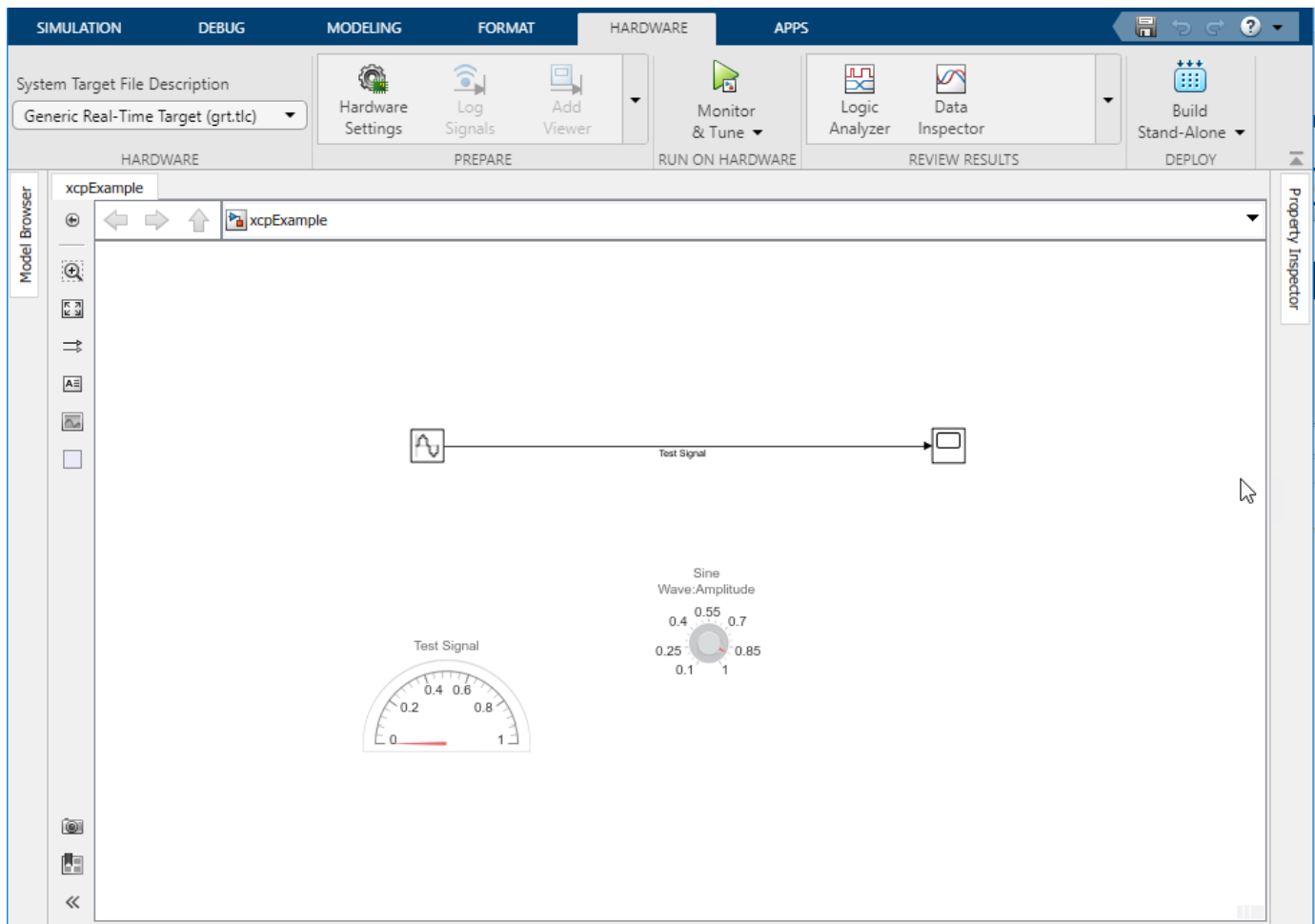
The Run on Custom Hardware app enables you to run external mode simulations on your development computer or other target hardware. In an external mode simulation, you can tune parameters in real time and monitor target application signals.

To run an external mode simulation, you:

- 1** Build the target application on your development computer.
- 2** Deploy the target application to the target hardware.
- 3** Connect Simulink to the target application that runs on the target hardware.
- 4** Start execution of generated code on the target hardware.

Using the app, you can:

- Perform the steps separately or with one click.
- Register custom launchers that deploy the target application.



Open the Run on Custom Hardware App

On the **Apps** tab, click **Run on Custom Hardware**. Or, on the **C Code** tab, select **Verify Code > Run on Custom Hardware**.

Examples

Run XCP External Mode Simulation

For an example that uses the Run on Custom Hardware app, see “Run XCP External Mode Simulation on Development Computer”.

Version History

Introduced in R2019b

Topics

“External Mode Simulation by Using XCP Communication”

“External Mode Simulation with TCP/IP or Serial Communication”

SIL/PIL Manager

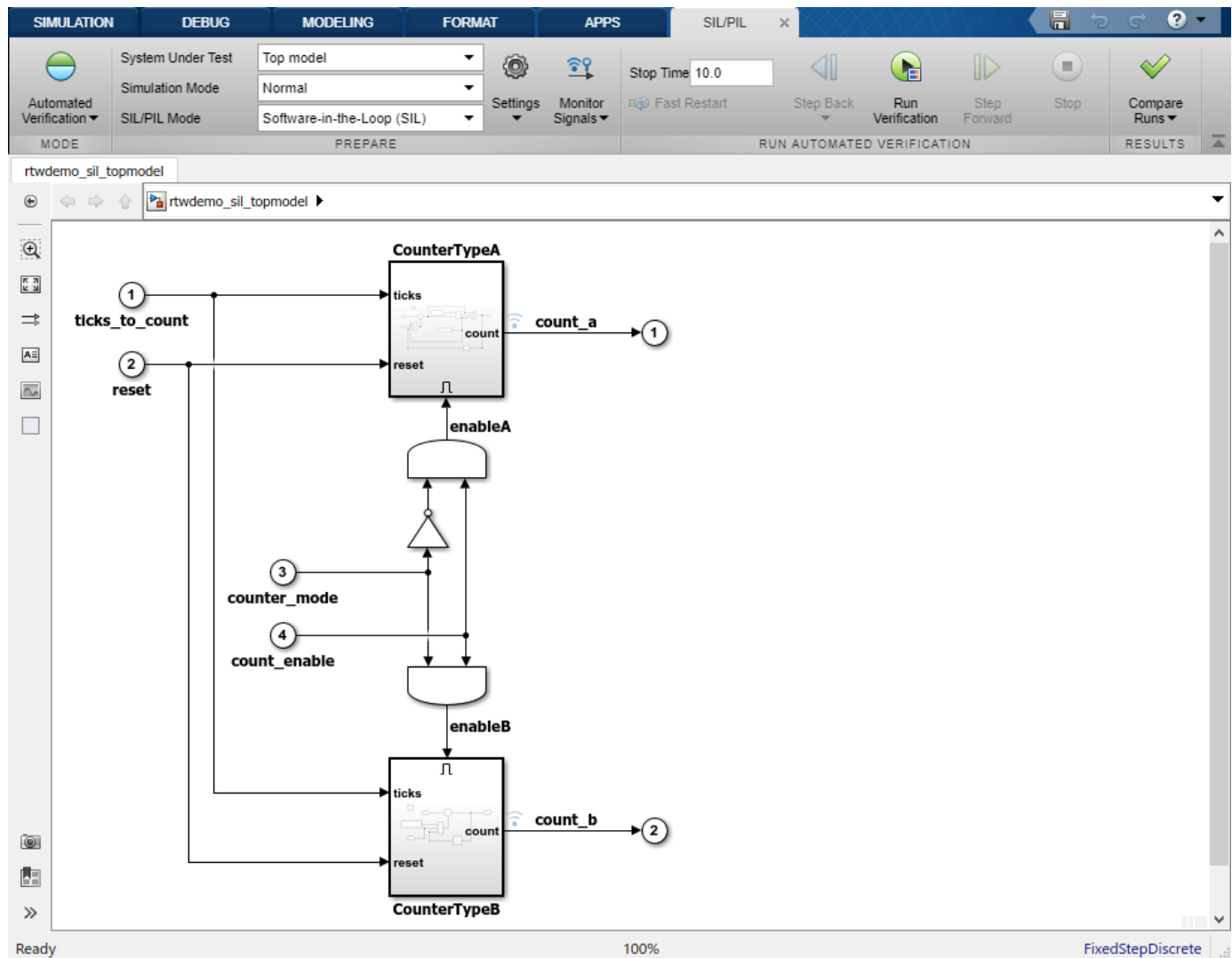
Verify generated code

Description

The SIL/PIL Manager simplifies verification of code that you generate from a model.

You can:

- With one click, test numeric equivalence between the model and generated code by running back-to-back model simulations and software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.
- Configure SIL or PIL simulations to produce code coverage, execution-time, and stack usage metrics.
- Enable your debugger for SIL simulations.
- Export automatically generated test cases for Simulink Test.



Open the SIL/PIL Manager App

On the Simulink toolstrip **Apps** tab, click **SIL/PIL Manager**. Or, on the Embedded Coder app **C Code** tab, click **Verify Code** > **SIL/PIL Manager**.

Examples

Verify Top-Model Code with a Single Click

- 1 In the Command Window, enter:

```
openExample('ecoder/SILPILVerificationExample', ...
    supportingFile='SILTopModel.slx')
```

The model opens in the Simulink Editor, with the **SIL/PIL** tab displayed.

- 2 SIL and PIL simulations do not support both code execution time profiling and code coverage analysis. For this example, disable code coverage:
 - 1 In the **Prepare** section, click **Settings**.
 - 2 Under **Coverage**, click the **Coverage Collection** button off.
- 3 In the **Run Automated Verification** section, click **Run Verification**.

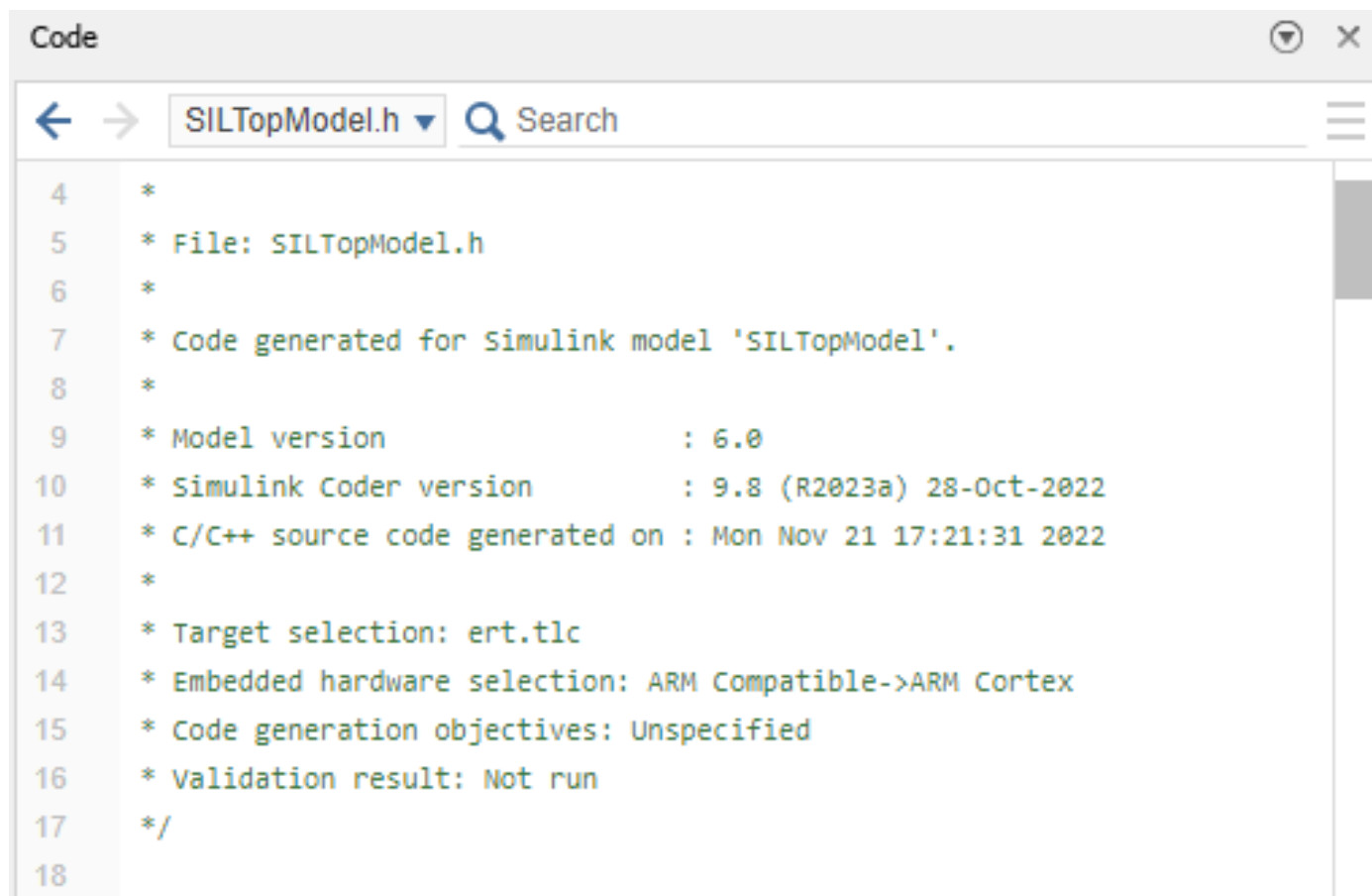
The SIL/PIL Manager runs these simulations back-to-back:

- SILTopModel in normal mode
- SILTopModel in SIL mode. As the **Time Profiling** controls are enabled, the SIL simulation also performs code execution profiling.

You can monitor simulation progress through the Diagnostic Viewer.

At the end of the second simulation:

- The SIL/PIL Manager displays generated code in the Code view, which enables you to analyze generated code, see code metrics, and trace between model elements and generated code.

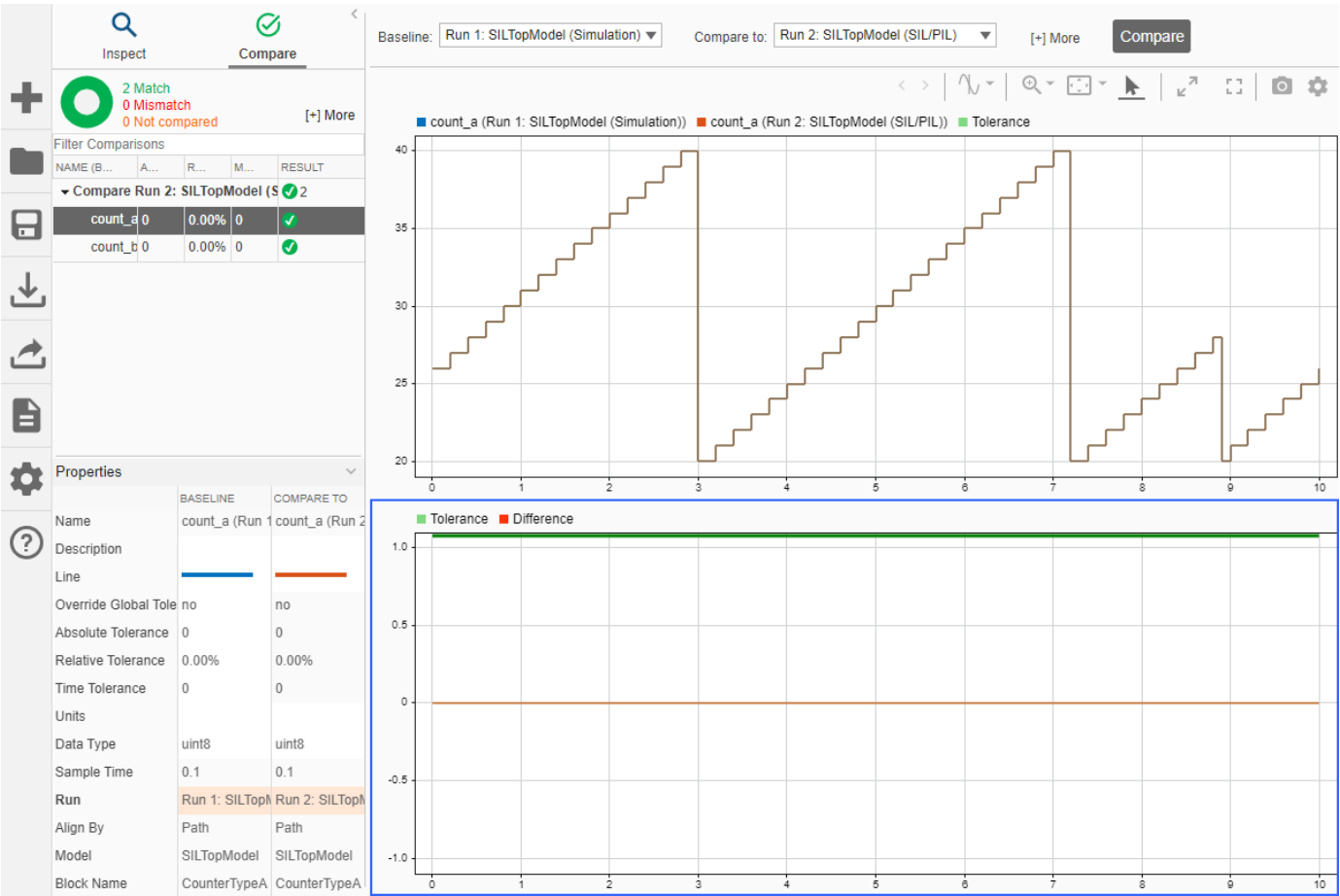


```

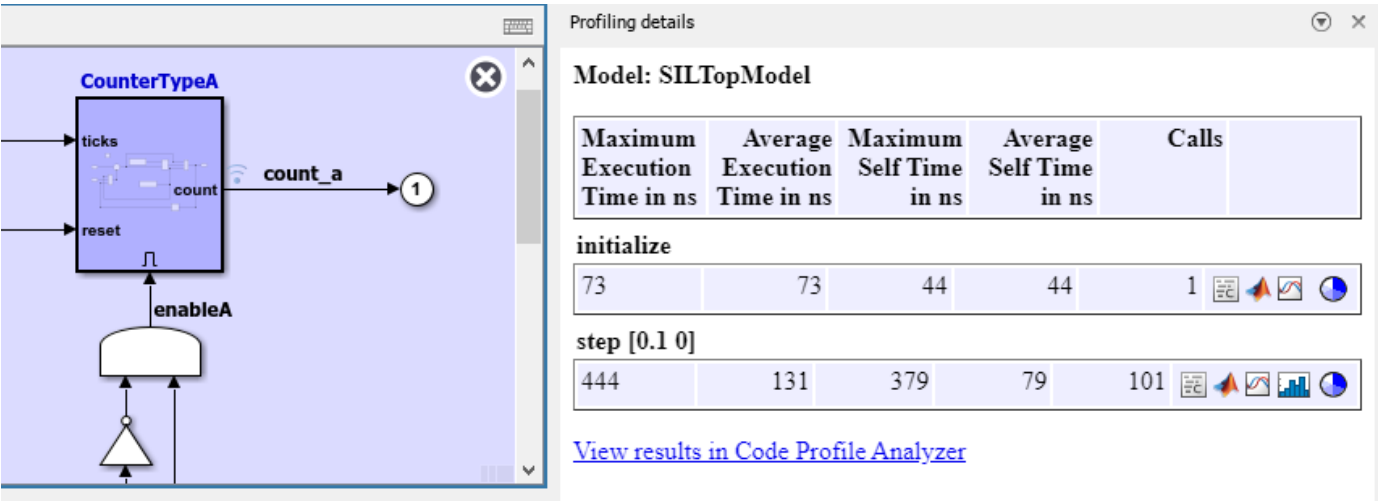
Code
< > SILTopModel.h Search
4  *
5  * File: SILTopModel.h
6  *
7  * Code generated for Simulink model 'SILTopModel'.
8  *
9  * Model version           : 6.0
10 * Simulink Coder version   : 9.8 (R2023a) 28-Oct-2022
11 * C/C++ source code generated on : Mon Nov 21 17:21:31 2022
12 *
13 * Target selection: ert.tlc
14 * Embedded hardware selection: ARM Compatible->ARM Cortex
15 * Code generation objectives: Unspecified
16 * Validation result: Not run
17 */
18

```

- The Simulation Data Inspector compares results from the model against results from generated code.



- To view execution-time metrics, in the model view, click the blue region.



To view the results in detail, open the Code Profile Analyzer by clicking the link.

Export Numeric Equivalence Test Case for Simulink Test

- 1 In the Command Window, enter:

```
openExample('ecoder/SILPILVerificationExample', ...  
           supportingFile='SILTopModel.slx')
```

The model opens in the Simulink Editor, with the **SIL/PIL** tab displayed.

- 2 For this example, disable code execution time profiling and code coverage:

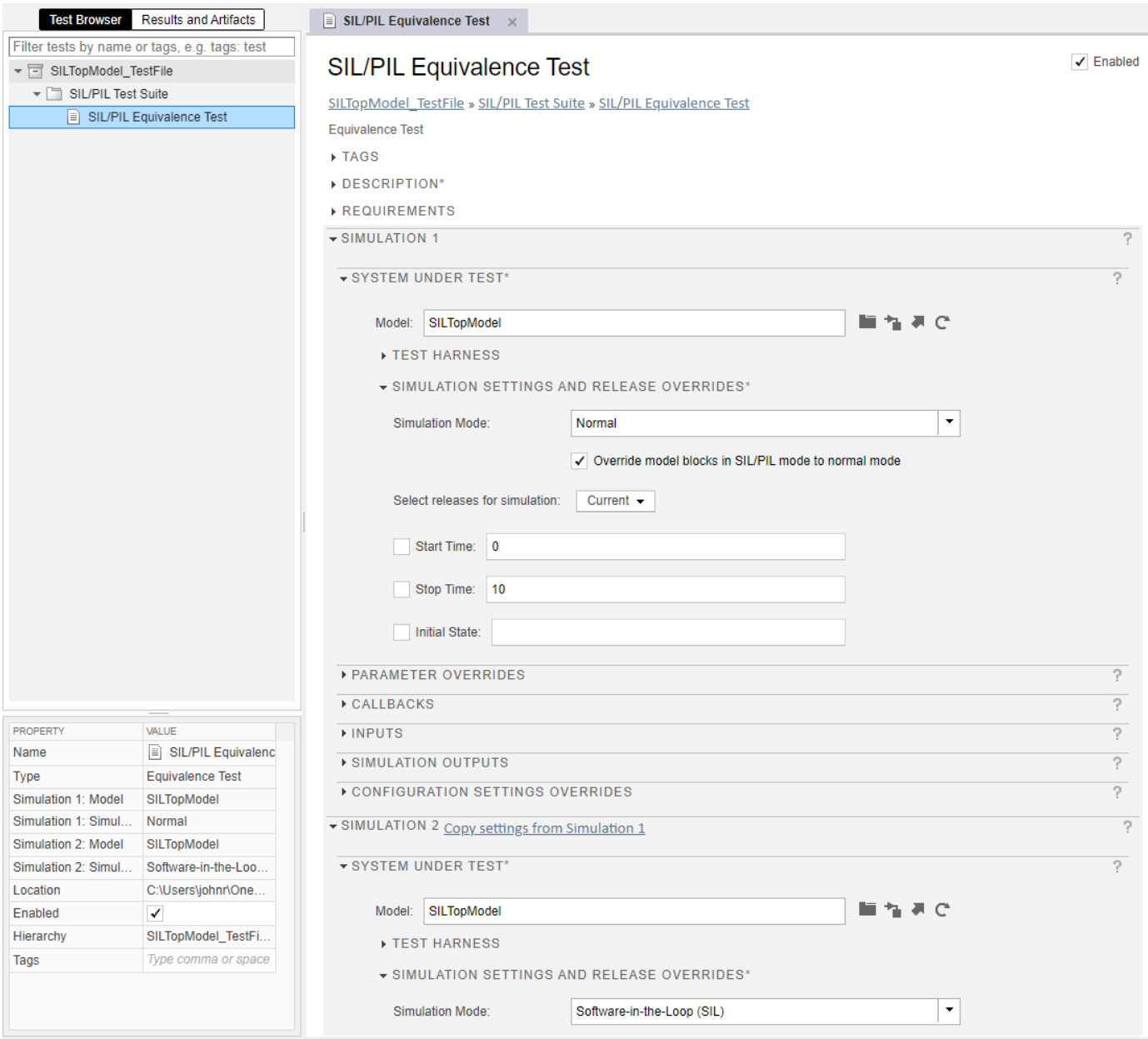
- 1 In the **Prepare** section, click **Settings**.
- 2 Under **Time Profiling**, click the **Task Profiling** button and the **Functions** button off.
- 3 Under **Coverage**, click the **Coverage Collection** button off.

- 3 At the end of the second simulation:

- Check that the results from the model and generated code are numerically equivalent. In the **Results** gallery, under **Numerical Equivalence Results**, click **Compare Runs** or **Data Inspector**.
- Export a test case for Simulink Test. Under **Simulink Test**, click **Export to Test Manager**. Then, in the Export SIL/PIL Test Cases dialog box, use the default settings, and click **OK**.

The SIL/PIL Manager:

- Creates the test case in `SILTopModel_TestFile.mldatx`, which you can find in the current working folder.
- Opens the test file that contains the test case.



For information about running test cases in Simulink Test, see “Import Test Cases for Equivalence Testing” (Simulink Test).

Generate Stack Usage Profile

- 1 In the Command Window, enter:

```
openExample('ecoder/SILPILVerificationExample', ...  
           supportingFile='SILTopModel.slx')
```

The model opens in the Simulink Editor, with the **SIL/PIL** tab displayed.

- 2 In the **Mode** section, select **SIL/PIL Simulation Only**.
- 3 In the **Prepare** section, open the **Settings** gallery:
 - 1 Under **Time Profiling**, click the **Task Profiling** button and **Functions** button off.
 - 2 Under **Coverage**, click the **Coverage Collection** button off.
 - 3 Under **Stack Profiling**, click the **Stack Profiling** button on.
- 4 In the **Run** section, click **Run SIL/PIL**.

The SIL simulation streams stack usage data to the Simulation Data Inspector. When the simulation is complete, the SIL/PIL Manager opens a code stack profiling report for the model.

Code Stack Profiling Report for SILTopModel

The code stack profiling report provides metrics based on data collected from a SIL or PIL execution. Stack usage is calculated from data recorded by instrumentation probes added to the SIL or PIL test harness and inside the code generated for each component. See [Code Stack Profiling](#) for more information.

1. Summary

Largest atomic integer type	Long
Word size (bits)	32
Pointer size (bits)	64
Stack driver	Target specific
Profiling data created	22-Nov-2022 10:54:29

2. Profiled Tasks

Task	Minimum Stack Usage in bytes	Average Stack Usage in bytes	Maximum Stack Usage in bytes	Minimum Function-Call Depth	Maximum Function-Call Depth	Calls	
initialize	96	96	96	2	2	1	
step [0.1 0]	112	112	112	2	2	101	 

3. Profiled Functions

Function	Memory in bytes	Calls	
CounterTypeA_Init	48	1	
CounterTypeA	48	101	
CounterTypeB_Init	48	1	
CounterTypeB	48	101	
step	64	101	
initialize	48	1	

- 5 To examine the results in detail, use the Code Profile Analyzer. In the **Results** gallery, under **Stack Profiling Results**, click the **Code Profile Analyzer** button.
 - “SIL/PIL Manager Verification Workflow”
 - “Choose a SIL or PIL Approach”
 - “View and Compare Code Execution Times”

- “Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode”
- “View and Compare Stack Usage Metrics”

Version History

Introduced in R2019b

See Also

Code Profile Analyzer

Topics

“SIL/PIL Manager Verification Workflow”

“Choose a SIL or PIL Approach”

“View and Compare Code Execution Times”

“Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode”

“View and Compare Stack Usage Metrics”