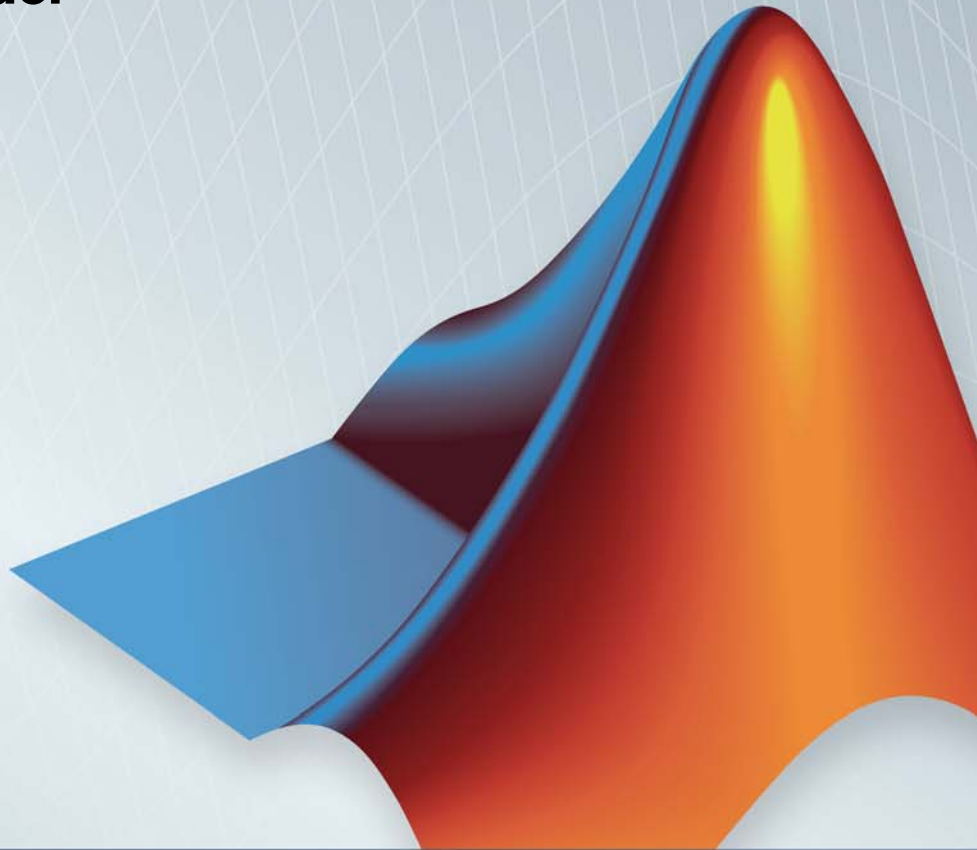


Simulink[®] Coder[™]

Reference

R2014a



MATLAB[®] & SIMULINK[®]



How to Contact MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Simulink[®] *Coder*[™] *Reference*

© COPYRIGHT 2011–2014 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

New for Version 8.0 (Release 2011a)
Revised for Version 8.1 (Release 2011b)
Revised for Version 8.2 (Release 2012a)
Revised for Version 8.3 (Release 2012b)
Revised for Version 8.4 (Release 2013a)
Revised for Version 8.5 (Release 2013b)
Revised for Version 8.6 (Release 2014a)

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/. Use the Saved Searches and Watched Bugs tool with the search phrase “Incorrect Code Generation” to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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.

Simulink Code Generation Limitations

1

Simulink Code Generation Limitations	1-2
--	-----

Alphabetical List

2

Blocks — Alphabetical List

3

Configuration Parameters for Simulink Models

4

Code Generation Pane: General	4-2
Code Generation: General Tab Overview	4-5
System target file	4-6
Browse	4-8
Language	4-9
Description	4-11
Target hardware	4-12
Toolchain	4-14
Build configuration	4-16
Tool/Options	4-18
Compiler optimization level	4-19
Custom compiler optimization flags	4-21
Generate makefile	4-22
Make command	4-24
Template makefile	4-26

Ignore custom storage classes	4-28
Ignore test point signals	4-30
Select objective	4-32
Prioritized objectives	4-34
Set Objectives	4-35
Set Objectives — Code Generation Advisor Dialog Box ...	4-36
Check Model	4-39
Check model before generating code	4-40
Generate code only	4-42
Build/Generate Code	4-44
Package code and artifacts	4-45
Zip file name	4-47
Code Generation Pane: Report	4-49
Code Generation: Report Tab Overview	4-51
Create code generation report	4-52
Open report automatically	4-55
Code-to-model	4-57
Model-to-code	4-59
Configure	4-61
Generate model Web view	4-62
Eliminated / virtual blocks	4-63
Traceable Simulink blocks	4-65
Traceable Stateflow objects	4-67
Traceable MATLAB functions	4-69
Static code metrics	4-71
Summarize which blocks triggered code replacements	4-73
Code Generation Pane: Comments	4-75
Code Generation: Comments Tab Overview	4-78
Include comments	4-79
Simulink block / Stateflow object comments	4-81
MATLAB source code as comments	4-82
Show eliminated blocks	4-84
Verbose comments for SimulinkGlobal storage class	4-85
Operator annotations	4-86
Simulink block descriptions	4-88
Simulink data object descriptions	4-90
Custom comments (MPT objects only)	4-92
Custom comments function	4-94
Stateflow object descriptions	4-96
Requirements in block comments	4-98
MATLAB function help text	4-100

Code Generation Pane: Symbols	4-102
Code Generation: Symbols Tab Overview	4-105
Global variables	4-106
Global types	4-109
Field name of global types	4-112
Subsystem methods	4-114
Subsystem method arguments	4-117
Local temporary variables	4-119
Local block output variables	4-122
Constant macros	4-124
Shared utilities	4-127
Minimum mangle length	4-128
Maximum identifier length	4-131
System-generated identifiers	4-133
Generate scalar inlined parameter as	4-138
Signal naming	4-139
M-function	4-141
Parameter naming	4-143
#define naming	4-145
Use the same reserved names as Simulation Target	4-147
Reserved names	4-148
Code Generation Pane: Custom Code	4-150
Code Generation: Custom Code Tab Overview	4-153
Use the same custom code settings as Simulation Target ..	4-154
Use local custom code settings (do not inherit from main model)	4-155
Source file	4-157
Header file	4-158
Initialize function	4-159
Terminate function	4-160
Include directories	4-161
Source files	4-163
Libraries	4-165
Code Generation Pane: Debug	4-167
Code Generation: Debug Tab Overview	4-169
Verbose build	4-170
Retain .rtw file	4-171
Profile TLC	4-172
Start TLC debugger when generating code	4-173
Start TLC coverage when generating code	4-175
Enable TLC assertion	4-176

Code Generation Pane: Interface	4-177
Code Generation: Interface Tab Overview	4-181
Standard math library	4-182
Code replacement library	4-184
Custom	4-187
Shared code placement	4-188
Support: floating-point numbers	4-190
Support: non-finite numbers	4-192
Support: complex numbers	4-194
Support: absolute time	4-195
Support: continuous time	4-197
Support: non-inlined S-functions	4-199
Support: variable-size signals	4-201
Multiword type definitions	4-202
Maximum word length	4-204
Code interface packaging	4-206
Multi-instance code error diagnostic	4-210
Pass root-level I/O as	4-212
Generate function to allocate model data	4-214
Classic call interface	4-216
Single output/update function	4-218
Terminate function required	4-221
Generate preprocessor conditionals	4-223
Suppress error status in real-time model data structure ..	4-225
Combine signal/state structures	4-227
Configure Model Functions	4-230
Block parameter visibility	4-231
Internal data visibility	4-233
Block parameter access	4-235
Internal data access	4-237
External I/O access	4-239
Generate destructor	4-241
Use operator new for referenced model object registration	4-243
Configure C++ Class Interface	4-245
MAT-file logging	4-246
MAT-file variable name modifier	4-249
Interface	4-251
Generate C API for: signals	4-254
Generate C API for: parameters	4-255
Generate C API for: states	4-256
Generate C API for: root-level I/O	4-257
Transport layer	4-258
MEX-file arguments	4-260

Static memory allocation	4-262
Static memory buffer size	4-264
Code Generation Pane: RSim Target	4-266
Code Generation: RSim Target Tab Overview	4-268
Enable RSim executable to load parameters from a MAT-file	4-269
Solver selection	4-270
Force storage classes to AUTO	4-271
Code Generation Pane: S-Function Target	4-272
Code Generation S-Function Target Tab Overview	4-274
Create new model	4-275
Use value for tunable parameters	4-276
Include custom source code	4-277
Code Generation Pane: Tornado Target	4-278
Code Generation: Tornado Target Tab Overview	4-280
Standard math library	4-281
Code replacement library	4-283
Shared code placement	4-285
MAT-file logging	4-287
MAT-file variable name modifier	4-289
Code Format	4-291
StethoScope	4-292
Download to VxWorks target	4-294
Base task priority	4-296
Task stack size	4-298
External mode	4-299
Transport layer	4-301
MEX-file arguments	4-303
Static memory allocation	4-305
Static memory buffer size	4-307
Code Generation: Coder Target Pane	4-309
Code Generation: Coder Target Pane Overview (previously “IDE Link Tab Overview”)	4-311
Coder Target: Tool Chain Automation Tab Overview	4-312
Build format	4-314
Build action	4-316
Overrun notification	4-319
Function name	4-321

Configuration	4-322
Compiler options string	4-324
Linker options string	4-326
System stack size (MAUs)	4-328
Profile real-time execution	4-331
Profile by	4-333
Number of profiling samples to collect	4-335
Maximum time allowed to build project (s)	4-337
Maximum time allowed to complete IDE operation (s)	4-339
Export IDE link handle to base workspace	4-340
IDE link handle name	4-342
Source file replacement	4-343
Parameter Reference	4-345
Recommended Settings Summary	4-345
Parameter Command-Line Information Summary	4-375

Model Advisor Checks

5

Simulink Coder Checks	5-2
Simulink Coder Checks Overview	5-3
Identify blocks using one-based indexing	5-4
Check solver for code generation	5-6
Check for blocks not supported by code generation	5-8
Check and update model to use toolchain approach to build generated code	5-9
Check and update the embedded target model to use ert.tlc system target file	5-12
Check for blocks that have constraints on tunable parameters	5-14
Check for model reference configuration mismatch	5-16
Check sample times and tasking mode	5-17
Code Generation Advisor Checks	5-17

Parameters for Creating Protected Models

6

Create Protected Model	6-2
Create Protected Model: Overview	6-3
Open read-only view of model	6-4
Simulate	6-5
Generate code	6-6
Generated code content type	6-7
Create protected model in	6-7
Create harness model for protected model	6-9

Simulink Code Generation Limitations

Simulink Code Generation Limitations

The following topics identify Simulink® code generation limitations:

- “C++ Language Limitations”
- “packNGo Function Limitations”
- “Tunable Expression Limitations”
- “Limitations on Data Type Specifications in Workspace”
- “Code Reuse Limitations for Subsystems”
- “Simulink Coder™ Model Referencing Limitations”
- “External Mode Limitations”
- “Noninlined S-Function Parameter Type Limitations”
- “S-Function Target Limitations”
- “Rapid Simulation Target Limitations”
- “Asynchronous Support Limitations”
- “C API Limitations”
- “Supported Products and Block Usage”

Alphabetical List

addCompileFlags

Purpose Add compiler options to model build information

Syntax `addCompileFlags(buildinfo, options, groups)`
groups is optional.

Arguments *buildinfo*
Build information returned by `RTW.BuildInfo`.

options
A character array or cell array of character arrays that specifies the compiler options to be added to the build information. The function adds each option to the end of a compiler option vector. If you specify multiple options within a single character array, for example `'-Zi -Wall'`, the function adds the string to the vector as a single element. For example, if you add `'-Zi -Wall'` and then `'-O3'`, the vector consists of two elements, as shown below.

```
'-Zi -Wall'    '-O3'
```

groups (optional)
A character array or cell array of character arrays that groups specified compiler options. You can use groups to

- Document the use of specific compiler options
- Retrieve or apply collections of compiler options

You can apply

- A single group name to one or more compiler options
- Multiple group names to collections of compiler options (available for non-makefile build environments only)

To...	Specify <i>groups</i> as a...
Apply one group name to one or more compiler options	Character array.
Apply different group names to compiler options	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>options</i> .

Note

- To specify compiler options to be used in the standard Simulink Coder makefile build process, specify *groups* as either 'OPTS' or 'OPT_OPTS'.
- To control compiler optimizations for your Simulink Coder makefile build at Simulink GUI level, use the **Compiler optimization level** parameter on the **Code Generation** pane of the Simulink Configuration Parameters dialog box. The **Compiler optimization level** parameter provides
 - Target-independent values `Optimizations on` (faster runs) and `Optimizations off` (faster builds), which allow you to easily toggle compiler optimizations on and off during code development
 - The value `Custom` for entering custom compiler optimization flags at Simulink GUI level (rather than at other levels of the build process)

If you use the configuration parameter **Make command** to specify compiler options for your Simulink Coder makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS="-v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

addCompileFlags

Description

The `addCompileFlags` function adds specified compiler options to the model build information. Simulink Coder stores the compiler options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *options* arguments, you can use an optional *groups* argument to group your options.

Examples

- Add the compiler option `-O3` to build information `myModelBuildInfo` and place the option in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;  
addCompileFlags(myModelBuildInfo, '-O3', 'OPTS');
```

- Add the compiler options `-Zi` and `-Wall` to build information `myModelBuildInfo` and place the options in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;  
addCompileFlags(myModelBuildInfo, '-Zi -Wall', 'OPT_OPTS');
```

- For a non-makefile build environment, add the compiler options `-Zi`, `-Wall`, and `-O3` to build information `myModelBuildInfo`. Place the options `-Zi` and `-Wall` in the group `Debug` and the option `-O3` in the group `MemOpt`.

```
myModelBuildInfo = RTW.BuildInfo;  
addCompileFlags(myModelBuildInfo, {'-Zi -Wall' '-O3'}, ...  
    {'Debug' 'MemOpt'});
```

See Also

`addDefines` | `addLinkFlags` | `getCompileFlags`

How To

- “Customize Post-Code-Generation Build Processing”

Purpose	Mark file, project, or build configuration as active
Syntax	<code>IDE_Obj.activate('objectname','type')</code>
IDEs	This function supports the following IDEs:
Description	<p>Use the <code>IDE_Obj.activate('objectname','type')</code> method to make a project file or build configuration active in the MATLAB® session.</p> <p>When you make a project, file, or build configuration active, methods you invoke on the IDE handle object apply to that project, file, or build configuration.</p>
Input Arguments	<p>IDE_Obj</p> <p>For <code>IDE_Obj</code>, enter the name of the IDE handle object you created using a constructor function.</p> <p>objectname</p> <p>For <code>objectname</code>, enter the name of the project file or build configuration to make active.</p> <p>For project files, enter the full file name including the extension.</p> <p>For build configurations, enter 'Debug', 'Release', or 'Custom'. Before using the <code>activate</code> method on a build configuration, activate the project that contains the build configuration. For more information about configurations, see “Configuration” on page 4-322.</p> <p>type</p> <p>For <code>type</code>, enter the type of object to make active. If you omit the <code>type</code> argument, <code>type</code> defaults to 'project'. Enter one of the following strings for <code>type</code>:</p> <ul style="list-style-type: none">• 'project' — Makes a specified project active.• 'buildcfg' — Make a specified build configuration active

activate

IDE support for *type*

	CCS	MULTI®	VisualDSP++®
'project'	Yes	Yes	Yes
'buildcfg'	Yes		Yes

Examples

After using a constructor to create the IDE handle object, h, open several projects, make the first one active, and build the project:

```
h.open('c:\temp\myproj1')
h.open('c:\temp\myproj2')
h.open('c:\temp\myproj3')
h.activate('c:\temp\myproj1', 'project')
h.build
```

After making a project active, make the 'debug' configuration active:

```
h.activate('debug', 'buildcfg')
```

See Also

build | new | remove

Purpose

Add files to active project in IDE

Syntax

`IDE_Obj.add(filename, filetype)`

IDEs

This function supports the following IDEs:

Description

Use `IDE_Obj.add(filename, filetype)` to add an existing file to the active project in the IDE. Using the add function is equivalent to selecting **Project > Add Files to Project** in the IDE.

Before using add:

- Use the constructor function for your IDE to create an IDE handle object, such as `IDE_Obj`.
- Create or open a project using the `new` or `open` methods.
- Make the project active in the IDE using the `activate` method.

You can add file types your IDE supports to your project. Consult the documentation for your IDE for detailed information about supported file types.

Supported File Types and Extensions

File Type	Extensions Supported	CCS IDE Project Folder
C/C++ source files	.c, .cpp, .cc, .cxx, .sa, .h, .hpp, .hxx	Source
Assembly source files	.a*, .s* (excluding .sa), .dsp	Source
Object and library files	.o*, .lib, .doj, .dlb	Libraries
Linker command file	.cmd, .ldf	Project Name

Supported File Types and Extensions (Continued)

File Type	Extensions Supported	CCS IDE Project Folder
VDK support file	.vdk	Not applicable
DSP/BIOS file (only with CCS IDE)	.tcf	DSP/BIOS Config

Note CCS IDE drops files in the project folder, indicated in the right-most column of the preceding table.

Input Arguments

`add` places the file specified by *filename* in the active project in the IDE.

IDE_Obj

IDE_Obj is a handle for an instance of the IDE. Before using a method, the constructor function for your IDE to create *IDE_Obj*.

filename

filename is the name of the file to add to the active IDE project.

If you supply a filename without a path or relative path, your coder product searches the IDE working folder first. It then searches the folders on your MATLAB path. Add supported file types shown in the preceding table.

filetype

filetype is an optional argument that specifies the file type. For example, 'lib', 'src', 'header'.

Examples

Start by creating an IDE handle object, such as `IDE_Obj` using the constructor for your IDE. Then enter the following commands:

```
IDE_Obj.new('myproject','project'); % Create a new project.
```

```
IDE_Obj.add('sourcefile.c'); % Add a C source file.
```

See Also

`activate` | `new` | `open` | `remove`

addDefines

Purpose Add preprocessor macro definitions to model build information

Syntax `addDefines(buildinfo, macrodefs, groups)`
groups is optional.

Arguments *buildinfo*
Build information returned by `RTW.BuildInfo`.

macrodefs
A character array or cell array of character arrays that specifies the preprocessor macro definitions to be added to the object. The function adds each definition to the end of a compiler option vector. If you specify multiple definitions within a single character array, for example `'-DRT -DDEBUG'`, the function adds the string to the vector as a single element. For example, if you add `'-DPROTO -DDEBUG'` and then `'-DPRODUCTION'`, the vector consists of two elements, as shown below.

```
'-DPROTO -DDEBUG'    '-DPRODUCTION'
```

groups (optional)
A character array or cell array of character arrays that groups specified definitions. You can use groups to

- Document the use of specific macro definitions
- Retrieve or apply groups of macro definitions

You can apply

- A single group name to one or more macro definitions
- Multiple group names to collections of macro definitions (available for non-makefile build environments only)

To...	Specify <i>groups</i> as a...
Apply one group name to one or more macro definitions	Character array.
Apply different group names to macro definitions	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>macrodefs</i> .

Note To specify macro definitions to be used in the standard Simulink Coder makefile build process, specify *groups* as either 'OPTS' or 'OPT_OPTS'.

Description

The `addDefines` function adds specified preprocessor macro definitions to the model build information. The Simulink Coder software stores the definitions in a vector. The function adds definitions to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *macrodefs* arguments, you can use an optional *groups* argument to group your options.

Examples

- Add the macro definition `-DPRODUCTION` to build information `myModelBuildInfo` and place the definition in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, '-DPRODUCTION', 'OPTS');
```

- Add the macro definitions `-DPROTO` and `-DDEBUG` to build information `myModelBuildInfo` and place the definitions in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
    '-DPROTO -DDEBUG', 'OPT_OPTS');
```

addDefines

- For a non-makefile build environment, add the macro definitions -DPROTO, -DDEBUG, and -DPRODUCTION to build information myModelBuildInfo. Place the definitions -DPROTO and -DDEBUG in the group Debug and the definition -DPRODUCTION in the group Release.

```
myModelBuildInfo = RTW.BuildInfo;  
addDefines(myModelBuildInfo, ...  
    {'-DPROTO -DDEBUG' '-DPRODUCTION'}, ...  
    {'Debug' 'Release'});
```

See Also

[addCompileFlags](#) | [addLinkFlags](#) | [getDefines](#)

How To

- “Customize Post-Code-Generation Build Processing”

Purpose

Add include files to model build information

Syntax

`addIncludeFiles(buildinfo, filenames, paths, groups)`

paths and *groups* are optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

filenames

A character array or cell array of character arrays that specifies names of include files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are `'*.*'`, `'*.h'`, and `'*.h*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate include file entries that

- You specify as input
- Already exist in the include file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

paths (optional)

A character array or cell array of character arrays that specifies paths to the include files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

groups (optional)

A character array or cell array of character arrays that groups specified include files. You can use groups to

addIncludeFiles

- Document the use of specific include files
- Retrieve or apply groups of include files

You can apply

- A single group name to an include file
- A single group name to multiple include files
- Multiple group names to collections of multiple include files

To...	Specify groups as a...
Apply one group name to one or more include files	Character array.
Apply different group names to include files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

Description

The `addIncludeFiles` function adds specified include files to the model build information. The Simulink Coder software stores the include files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a... The Function...

Character array	Applies the character array to include files it adds to the build information
Cell array of character arrays	Pairs each character array with a specified include file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string (' ') for *paths*.

Note The `packNGo` function also can add include files to the model build information. If you call the `packNGo` function to package model code, `packNGo` finds include files from source and include paths recorded in the model build information and adds them to the build information.

Examples

- Add the include file `mytypes.h` to build information `myModelBuildInfo` and place the file in the group `SysFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
    'mytypes.h', '/proj/src', 'SysFiles');
```

- Add the include files `etc.h` and `etc_private.h` to build information `myModelBuildInfo` and place the files in the group `AppFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
    {'etc.h' 'etc_private.h'}, ...
    '/proj/src', 'AppFiles');
```

- Add the include files `etc.h`, `etc_private.h`, and `mytypes.h` to build information `myModelBuildInfo`. Group the files `etc.h` and

addIncludeFiles

etc_private.h with the string AppFiles and the file mytypes.h with the string SysFiles.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
{'etc.h' 'etc_private.h' 'mytypes.h'}, ...
'/proj/src', ...
{'AppFiles' 'AppFiles' 'SysFiles'});
```

- Add the .h files in a specified folder to build information myModelBuildInfo and place the files in the group HFiles.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, ...
'*.h', '/proj/src', 'HFiles');
```

See Also

[addIncludePaths](#) | [addSourceFiles](#) | [addSourcePaths](#)
| [findIncludeFiles](#) | [getIncludeFiles](#) |
[updateFilePathsAndExtensions](#) | [updateFileSeparator](#)

How To

- “Customize Post-Code-Generation Build Processing”

Purpose

Add include paths to model build information

Syntax

`addIncludePaths(buildinfo, paths, groups)`

`groups` is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

paths

A character array or cell array of character arrays that specifies include file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

The function removes duplicate include file entries that

- You specify as input
- Already exist in the include path vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

groups (optional)

A character array or cell array of character arrays that groups specified include paths. You can use groups to

- Document the use of specific include paths
- Retrieve or apply groups of include paths

You can apply

- A single group name to an include path
- A single group name to multiple include paths
- Multiple group names to collections of multiple include paths

addIncludePaths

To...	Specify groups as a...
Apply one group name to one or more include paths	Character array.
Apply different group names to include paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>paths</i> .

Description

The `addIncludePaths` function adds specified include paths to the model build information. The Simulink Coder software stores the include paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *paths* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to include paths it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified include path. Thus, the length of the cell array must match the length of the cell array you specify for <i>paths</i> .

Examples

- Add the include path `/etcproj/etc/etc_build` to build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
    '/etcproj/etc/etc_build');
```

- Add the include paths `/etcproj/etclib` and `/etcproj/etc/etc_build` to build information `myModelBuildInfo` and place the files in the group `etc`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
    {'/etcproj/etclib' '/etcproj/etc/etc_build'}, 'etc');
```

- Add the include paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to build information `myModelBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/etc_build` with the string `etc` and the path `/common/lib` with the string `shared`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
    {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
    '/common/lib'}, {'etc' 'etc' 'shared'});
```

See Also

[addIncludeFiles](#) | [addSourceFiles](#) | [addSourcePaths](#)
| [getIncludePaths](#) | [updateFilePathsAndExtensions](#) |
[updateFileSeparator](#)

How To

- “Customize Post-Code-Generation Build Processing”

addLinkFlags

Purpose Add link options to model build information

Syntax `addLinkFlags(buildinfo, options, groups)`
groups is optional.

Arguments *buildinfo*
Build information returned by `RTW.BuildInfo`.

options
A character array or cell array of character arrays that specifies the linker options to be added to the build information. The function adds each option to the end of a linker option vector. If you specify multiple options within a single character array, for example `'-MD -Gy'`, the function adds the string to the vector as a single element. For example, if you add `'-MD -Gy'` and then `'-T'`, the vector consists of two elements, as shown below.

```
'-MD -Gy'    '-T'
```

groups (optional)
A character array or cell array of character arrays that groups specified linker options. You can use groups to

- Document the use of specific linker options
- Retrieve or apply groups of linker options

You can apply

- A single group name to one or more linker options
- Multiple group names to collections of linker options (available for non-makefile build environments only)

To...	Specify <i>groups</i> as a...
Apply one group name to one or more linker options	Character array.
Apply different group names to linker options	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>options</i> .

Note To specify linker options to be used in the standard Simulink Coder makefile build process, specify *groups* as either 'OPTS' or 'OPT_OPTS'.

Description

The `addLinkFlags` function adds specified linker options to the model build information. The Simulink Coder software stores the linker options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

In addition to the required *buildinfo* and *options* arguments, you can use an optional *groups* argument to group your options.

Examples

- Add the linker -T option to build information `myModelBuildInfo` and place the option in the group `OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, '-T', 'OPTS');
```

- Add the linker options -MD and -Gy to build information `myModelBuildInfo` and place the options in the group `OPT_OPTS`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, '-MD -Gy', 'OPT_OPTS');
```

addLinkFlags

- For a non-makefile build environment, add the linker options `-MD`, `-Gy`, and `-T` to build information `myModelBuildInfo`. Place the options `-MD` and `-Gy` in the group `Debug` and the option `-T` in the group `Temp`.

```
myModelBuildInfo = RTW.BuildInfo;  
addLinkFlags(myModelBuildInfo, {'-MD -Gy' '-T'}, ...  
    {'Debug' 'Temp'});
```

See Also

`addCompileFlags` | `addDefines` | `getLinkFlags`

How To

- “Customize Post-Code-Generation Build Processing”

Purpose

Add link objects to model build information

Syntax

```
addLinkObjects(buildinfo, linkobjs, paths, priority,  
precompiled, linkonly, groups)
```

Arguments except *buildinfo*, *linkobjs*, and *paths* are optional. If you specify an optional argument, you must specify the optional arguments preceding it.

Arguments

buildinfo

Build information returned by RTW.BuildInfo.

linkobjs

A character array or cell array of character arrays that specifies the filenames of linkable objects to be added to the build information. The function adds the filenames that you specify in the function call to a vector that stores the object filenames in priority order. If you specify multiple objects that have the same priority (see *priority* below), the function adds them to the vector based on the order in which you specify the object filenames in the cell array.

The function removes duplicate link objects that

- You specify as input
- Already exist in the linkable object filename vector
- Have a path that matches the path of a matching linkable object filename

A duplicate entry consists of an exact match of a path string and corresponding linkable object filename.

paths

A character array or cell array of character arrays that specifies paths to the linkable objects. If you specify a character array, the path string applies to all linkable objects.

addLinkObjects

priority (optional)

A numeric value or vector of numeric values that indicates the relative priority of each specified link object. Lower values have higher priority. The default priority is 1000.

precompiled (optional)

The logical value `true` or `false`, or a vector of logical values that indicates whether each specified link object is precompiled.

Specify `true` if the link object has been prebuilt for faster compiling and linking and exists in a specified location.

If `precompiled` is `false` (the default), the Simulink Coder build process creates the link object in the build folder.

This argument is ignored if *linkonly* equals `true`.

linkonly (optional)

The logical value `true` or `false`, or a vector of logical values that indicates whether each specified link object is to be used only for linking.

Specify `true` if the Simulink Coder build process should not build, nor generate rules in the makefile for building, the specified link object, but should include it when linking the final executable. For example, you can use this to incorporate link objects for which source files are not available. If *linkonly* is `true`, the value of *precompiled* is ignored.

If *linkonly* is `false` (the default), rules for building the link objects are added to the makefile. In this case, the value of *precompiled* determines which subsection of the added rules is expanded, `START_PRECOMP_LIBRARIES` (`true`) or `START_EXPAND_LIBRARIES` (`false`).

groups (optional)

A character array or cell array of character arrays that groups specified link objects. You can use groups to

- Document the use of specific link objects
- Retrieve or apply groups of link objects

You can apply

- A single group name to a linkable object
- A single group name to multiple linkable objects
- Multiple group name to collections of multiple linkable objects

To...	Specify groups as a...
Apply one group name to one or more link objects	Character array.
Apply different group names to link objects	Cell array of character arrays such that the number of group names matches the number of elements you specify for <i>linkobjs</i> .

The default value of *groups* is { ' ' }.

Description

The `addLinkObjects` function adds specified link objects to the model build information. The Simulink Coder software stores the link objects in a vector in relative priority order. If multiple objects have the same priority or you do not specify priorities, the function adds the objects to the vector based on the order in which you specify them.

In addition to the required *buildinfo*, *linkobjs*, and *paths* arguments, you can specify the optional arguments *priority*, *precompiled*, *linkonly*, and *groups*. You can specify *paths* and *groups* as a character array or a cell array of character arrays.

addLinkObjects

If You Specify <i>paths</i> or <i>groups</i> as a...	The Function...
Character array	Applies the character array to objects it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified object. Thus, the length of the cell array must match the length of the cell array you specify for <i>linkobjs</i> .

Similarly, you can specify *priority*, *precompiled*, and *linkonly* as a value or vector of values.

If You Specify <i>priority</i>, <i>precompiled</i>, or <i>linkonly</i> as a...	The Function...
Value	Applies the value to objects it adds to the build information.
Vector of values	Pairs each value with a specified object. Thus, the length of the vector must match the length of the cell array you specify for <i>linkobjs</i> .

If you choose to specify an optional argument, you must specify optional arguments preceding it. For example, to specify that objects are precompiled using the *precompiled* argument, you must specify the *priority* argument that precedes *precompiled*. You could pass the default priority value 1000, as shown below.

```
addLinkObjects(myBuildInfo, 'test1', '/proj/lib/lib1', 1000, true);
```

Examples

- Add the linkable objects `libobj1` and `libobj2` to build information `myModelBuildInfo` and set the priorities of the objects to 26 and 10, respectively. Since `libobj2` is assigned the lower numeric priority

value, and thus has the higher priority, the function orders the objects such that `libobj2` precedes `libobj1` in the vector.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo, {'libobj1' 'libobj2'},...
{'/proj/lib/lib1' '/proj/lib/lib2'}, [26 10]);
```

- Add the linkable objects `libobj1` and `libobj2` to build information `myModelBuildInfo`. Mark both objects as link-only. Since individual priorities are not specified, the function adds the objects to the vector in the order specified.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo, {'libobj1' 'libobj2'},...
{'/proj/lib/lib1' '/proj/lib/lib2'}, 1000,...
false, true);
```

- Add the linkable objects `libobj1` and `libobj2` to build information `myModelBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Mark both objects as precompiled, and group them under the name `MyTest`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo, {'libobj1' 'libobj2'},...
{'/proj/lib/lib1' '/proj/lib/lib2'}, [26 10],...
true, false, 'MyTest');
```

How To

- “Customize Post-Code-Generation Build Processing”

addNonBuildFiles

Purpose

Add nonbuild-related files to model build information

Syntax

`addNonBuildFiles(buildinfo, filenames, paths, groups)`
paths and *groups* are optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

filenames

A character array or cell array of character arrays that specifies names of nonbuild-related files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are `'*.*'`, `'*.DLL'`, and `'*.D*'`.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate nonbuild file entries that

- Already exist in the nonbuild file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

paths (optional)

A character array or cell array of character arrays that specifies paths to the nonbuild files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

groups (optional)

A character array or cell array of character arrays that groups specified nonbuild files. You can use groups to

- Document the use of specific nonbuild files
- Retrieve or apply groups of nonbuild files

You can apply

- A single group name to a nonbuild file
- A single group name to multiple nonbuild files
- Multiple group names to collections of multiple nonbuild files

To...	Specify groups as a...
Apply one group name to one or more nonbuild files	Character array.
Apply different group names to nonbuild files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

Description

The `addNonBuildFiles` function adds specified nonbuild-related files, such as DLL files required for a final executable, or a README file, to the model build information. The Simulink Coder software stores the nonbuild files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

addNonBuildFiles

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to nonbuild files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified nonbuild file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string (' ') for *paths*.

Examples

- Add the nonbuild file `readme.txt` to build information `myModelBuildInfo` and place the file in the group `DocFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
    'readme.txt', '/proj/docs', 'DocFiles');
```

- Add the nonbuild files `myutility1.dll` and `myutility2.dll` to build information `myModelBuildInfo` and place the files in the group `DLLFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
    {'myutility1.dll' 'myutility2.dll'}, ...
    '/proj/dlls', 'DLLFiles');
```

- Add the DLL files in a specified folder to build information `myModelBuildInfo` and place the files in the group `DLLFiles`.

```
myModelBuildInfo = RTW.BuildInfo;
addNonBuildFiles(myModelBuildInfo, ...
    '*.dll', '/proj/dlls', 'DLLFiles');
```

See Also

`getNonBuildFiles`

How To

- “Customize Post-Code-Generation Build Processing”

Purpose

Memory address and page value of symbol in IDE

Syntax

```
a = IDE_Obj.address(symbol,scope)
```

IDEs

This function supports the following IDEs:

Description

The `a = IDE_Obj.address(symbol,scope)` method returns the memory address of the first matching symbol in the symbol table of the most recently loaded program.

Because the `address` method returns the `address` and `page` values as a structure, your programs can use the values directly. For example, the `IDE_Obj.read` and `IDE_Obj.write` can use `a` as an input.

If the `address` method does not find the symbol in the symbol table, it generates a warning and returns a null value.

Input Arguments***a***

Use `a` as a variable to capture the return values from the `address` method.

IDE_Obj

`IDE_Obj` is a handle for an instance of the IDE. Before using a method, use the constructor function for your IDE to create `IDE_Obj`.

symbol

`symbol` is the name of the symbol for which you are getting the memory address and page values.

Symbol names are case sensitive.

For `address` to return an address, the symbol must be a valid entry in the symbol table. If the `address` method does not find the symbol, it generates a warning and leaves a empty.

scope

address

Optionally, you set the scope of the address method. Enter 'local' or 'global'. Use 'local' when the current scope of the program is the desired function scope. If you omit the *scope* argument, the address method uses 'local' by default.

Output Arguments

If the address method does not find the symbol, it generates a warning and does not return a value for *a*.

The address method only returns address information for the first matching symbol in the symbol table.

For Code Composer Studio™

The return value, *a*, is a numeric array with the symbol's address offset, *a*(1), and page, *a*(2).

With TI C6000™ processors, the memory page value is 0.

For MULTI

With MULTI, address requires a linker command file (*lcf*) in your project.

The return value, *a*, is a numeric array with the symbol's address offset, *a*(1), and page, *a*(2).

For VisualDSP++

With VisualDSP++, address requires a linker command file (*lcf*) in your project.

The return value *a* is a numeric array with the symbol's start address, *a*(1), and memory type, *a*(2).

Examples

After you load a program to your processor, address lets you read and write to specific entries in the symbol table for the program. For example, the following function reads the value of symbol 'ddat' from the symbol table in the IDE.

```
ddatv = IDE_Obj.read(IDE_Obj.address('ddat'),'double',4)
```

ddat is an entry in the current symbol table. `address` searches for the string *ddat* and returns a value when it finds a match. `read` returns *ddat* to MATLAB software as a double-precision value as specified by the string 'double'.

To change values in the symbol table, use `address` with `write`:

```
IDE_Obj.write(IDE_Obj.address('ddat'),double([pi 12.3 exp(-1)...  
sin(pi/4)]))
```

After executing this write operation, *ddat* contains double-precision values for π , 12.3, e^{-1} , and $\sin(\pi/4)$. Use `read` to verify the contents of *ddat*:

```
ddatv = IDE_Obj.read(IDE_Obj.address('ddat'),'double',4)
```

MATLAB software returns

```
ddatv =
```

```
    3.1416    12.3    0.3679    0.7071
```

See Also

`load` | `read` | `write`

addSourceFiles

Purpose Add source files to model build information

Syntax `addSourceFiles(buildinfo, filenames, paths, groups)`
paths and *groups* are optional.

Arguments *buildinfo*
Build information returned by `RTW.BuildInfo`.

filenames
A character array or cell array of character arrays that specifies names of the source files to be added to the build information.

The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are '*.*', '*.c', and '*.c*'.

The function adds the filenames to the end of a vector in the order that you specify them.

The function removes duplicate source file entries that

- You specify as input
- Already exist in the source file vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

paths (optional)
A character array or cell array of character arrays that specifies paths to the source files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.

groups (optional)
A character array or cell array of character arrays that groups specified source files. You can use groups to

- Document the use of specific source files
- Retrieve or apply groups of source files

You can apply

- A single group name to a source file
- A single group name to multiple source files
- Multiple group names to collections of multiple source files

To...	Specify group as a...
Apply one group name to one or more source files	Character array.
Apply different group names to source files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> .

Description

The `addSourceFiles` function adds specified source files to the model build information. The Simulink Coder software stores the source files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *filenames* arguments, you can specify optional *paths* and *groups* arguments. You can specify each optional argument as a character array or a cell array of character arrays.

addSourceFiles

If You Specify an Optional Argument as a... The Function...

Character array	Applies the character array to source files it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source file. Thus, the length of the cell array must match the length of the cell array you specify for <i>filenames</i> .

If you choose to specify *groups*, but omit *paths*, specify a null string (' ') for *paths*.

Examples

- Add the source file `driver.c` to build information `myModelBuildInfo` and place the file in the group `Drivers`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, 'driver.c', ...
'/proj/src', 'Drivers');
```

- Add the source files `test1.c` and `test2.c` to build information `myModelBuildInfo` and place the files in the group `Tests`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
{'test1.c' 'test2.c'}, ...
'/proj/src', 'Tests');
```

- Add the source files `test1.c`, `test2.c`, and `driver.c` to build information `myModelBuildInfo`. Group the files `test1.c` and `test2.c` with the string `Tests` and the file `driver.c` with the string `Drivers`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, ...
{'test1.c' 'test2.c' 'driver.c'}, ...
'/proj/src', ...
{'Tests' 'Tests' 'Drivers'});
```

- Add the .c files in a specified folder to build information myModelBuildInfo and place the files in the group CFiles.

```
myModelBuildInfo = RTW.BuildInfo;  
addIncludeFiles(myModelBuildInfo, ...  
    '*.c', '/proj/src', 'CFiles');
```

See Also

[addIncludeFiles](#) | [addIncludePaths](#) | [addSourcePaths](#)
| [getSourceFiles](#) | [updateFilePathsAndExtensions](#) |
[updateFileSeparator](#)

How To

- “Customize Post-Code-Generation Build Processing”

addSourcePaths

Purpose Add source paths to model build information

Syntax `addSourcePaths(buildinfo, paths, groups)`
groups is optional.

Arguments *buildinfo*
Build information returned by `RTW.BuildInfo`.

paths
A character array or cell array of character arrays that specifies source file paths to be added to the build information. The function adds the paths to the end of a vector in the order that you specify them.

The function removes duplicate source file entries that

- You specify as input
- Already exist in the source path vector
- Have a path that matches the path of a matching filename

A duplicate entry consists of an exact match of a path string and corresponding filename.

Note The Simulink Coder software does not check whether a specified path string is valid.

groups (optional)
A character array or cell array of character arrays that groups specified source paths. You can use groups to

- Document the use of specific source paths
- Retrieve or apply groups of source paths

You can apply

- A single group name to a source path
- A single group name to multiple source paths
- Multiple group names to collections of multiple source paths

To...	Specify groups as a...
Apply one group name to one or more source paths	Character array.
Apply different group names to source paths	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>paths</i> .

Description

The `addSourcePaths` function adds specified source paths to the model build information. The Simulink Coder software stores the source paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

In addition to the required *buildinfo* and *paths* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to source paths it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified source path. Thus, the length of the character array or cell array must match the length of the cell array you specify for <i>paths</i> .

addSourcePaths

Note The Simulink Coder software does not check whether a specified path string is valid.

Examples

- Add the source path `/etcproj/etc/etc_build` to build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,...
    '/etcproj/etc/etc_build');
```

- Add the source paths `/etcproj/etclib` and `/etcproj/etc/etc_build` to build information `myModelBuildInfo` and place the files in the group `etc`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,...
    {'/etcproj/etclib' '/etcproj/etc/etc_build'}, 'etc');
```

- Add the source paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to build information `myModelBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/etc_build` with the string `etc` and the path `/common/lib` with the string `shared`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,...
    {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
    '/common/lib'}, {'etc' 'etc' 'shared'});
```

See Also

[addIncludeFiles](#) | [addIncludePaths](#) | [addSourceFiles](#)
| [getSourcePaths](#) | [updateFilePathsAndExtensions](#) |
[updateFileSeparator](#)

How To

- “Customize Post-Code-Generation Build Processing”

Purpose

Add template makefile (TMF) tokens that provide build-time information for makefile generation

Syntax

`addTMFTokens(buildinfo, tokennames, tokenvalues, groups)`
groups is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

tokennames

A character array or cell array of character arrays that specifies names of TMF tokens (for example, '|>CUSTOM_OUTNAME<|') to be added to the build information. The function adds the token names to the end of a vector in the order that you specify them.

If you specify a token name that already exists in the vector, the first instance takes precedence and its value is used for replacement.

tokenvalues

A character array or cell array of character arrays that specifies TMF token values corresponding to the previously-specified TMF token names. The function adds the token values to the end of a vector in the order that you specify them.

groups (optional)

A character array or cell array of character arrays that groups specified TMF tokens. You can use groups to

- Document the use of specific TMF tokens
- Retrieve or apply groups of TMF tokens

You can apply

- A single group name to a TMF token
- A single group name to multiple TMF tokens
- Multiple group names to collections of multiple TMF tokens

addTMFTokens

To...	Specify groups as a...
Apply one group name to one or more TMF tokens	Character array.
Apply different group names to TMF tokens	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>tokennames</i> .

Description

Call the `addTMFTokens` function inside a post code generation command to provide build-time information to help customize makefile generation. The tokens specified in the `addTMFTokens` function call must be handled in the template makefile (TMF) for the target selected for your model. For example, if your post code generation command calls `addTMFTokens` to add a TMF token named `|>CUSTOM_OUTNAME<|` that specifies an output file name for the build, the TMF must take action with the value of `|>CUSTOM_OUTNAME<|` to achieve the desired result. (See “Examples” on page 2-43.)

The `addTMFTokens` function adds specified TMF token names and values to the model build information. The Simulink Coder software 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.

In addition to the required *buildinfo*, *tokennames*, and *tokenvalues* arguments, you can specify an optional *groups* argument. You can specify *groups* as a character array or a cell array of character arrays.

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to TMF tokens it adds to the build information.
Cell array of character arrays	Pairs each character array with a specified TMF token. Thus, the length of the cell array must match the length of the cell array you specify for <i>tokennames</i> .

Examples

Inside a post code generation command, add the TMF token `|>CUSTOM_OUTNAME<|` and its value to build information `myModelBuildInfo`, and place the token in the group `LINK_INFO`.

```
myModelBuildInfo = RTW.BuildInfo;
addTMFTokens(myModelBuildInfo, ...
             '|>CUSTOM_OUTNAME<|', 'foo.exe', 'LINK_INFO');
```

In the TMF for the target selected for your model, code such as the following uses the token value to achieve the desired result:

```
CUSTOM_OUTNAME = |>CUSTOM_OUTNAME<|
...
target:
$(LD) -o $(CUSTOM_OUTNAME) ...
```

How To

- “Customize Post-Code-Generation Build Processing”

build

Purpose Build or rebuild current project

Syntax `[result,numwarns]=IDE_Obj.build(timeout)`
`IDE_Obj.build('all')`

IDEs This function supports the following IDEs:

Description `[result,numwarns]=IDE_Obj.build(timeout)` incrementally builds the active project. Incremental builds recompile only source files in your project that you changed or added after the most recent build. `build` uses the file time stamp to determine whether to recompile a file. After recompiling the source files, `build` links the object files to make a new program file.

The value of `result` is 1 when the build process completes. The value of `numwarns` is the number of compilation warnings generated from the build process.

The *timeout* argument defines the number of seconds MATLAB waits for the IDE to complete the build process. If the IDE exceeds the timeout period, this method returns a timeout error immediately. The timeout error does not terminate the build process in the IDE. The IDE continues the build process. The timeout error indicates that the build process did not complete before the specified timeout period expired. If you omit the *timeout* argument, the build method uses a default value of 1000 seconds.

`IDE_Obj.build('all')` rebuilds the files in the active project.

See Also `isrunning` | `open`

Purpose	Close project in IDE window
Syntax	<code>IDE_Obj.close(filename, 'project')</code>
IDEs	This function supports the following IDEs:
Description	<p>Use <code>IDE_Obj.close(filename, 'project')</code> to close a specific project, projects, or the active open project.</p> <p>For the <i>filename</i> argument:</p> <ul style="list-style-type: none">• To close the project files, enter 'all'.• To close a specific project, enter the project file name, such as 'myProj'. If the file is not an open file in the IDE, MATLAB returns a warning message.• To close the active project, enter []. <p>With the VisualDSP++ IDE, to close the current project group (if <i>filename</i> is 'all' or []), replace 'project' with 'projectgroup'.</p> <hr/> <p>Note</p> <ul style="list-style-type: none">• The open method does not support the 'text' argument.• Save changes to your files and projects in the IDE before you use <code>close</code>. The <code>close</code> method does not save changes, nor does it prompt you to save changes, before it closes the project. <hr/>
Examples	<p>To close the open project files:</p> <pre>IDE_Obj.close('all', 'project')</pre> <p>To close the open project, myProj:</p> <pre>IDE_Obj.close('myProj', 'project')</pre>

close

To close the active open project:

```
IDE_Obj.close([], 'project')
```

With the VisualDSP++ IDE, to close the open project groups:

```
IDE_Obj.close('all', 'projectgroup')
```

With the VisualDSP++ IDE, to close the active project group:

```
IDE_Obj.close([], 'projectgroup')
```

See Also

add | open

Purpose Close HTML code generation report

Syntax `coder.report.close()`

Description `coder.report.close()` closes the HTML code generation report.

Examples **Close code generation report for a model**

After opening a code generation report for `rtwdemo_counter`, close the report.

```
coder.report.close()
```

See Also `coder.report.open` | `coder.report.generate`

Concepts

- “Reports for Code Generation”

coder.report.generate

Purpose 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.

Input Arguments

model - Model name

string

Model name specified as a string

Example: 'rtwdemo_counter'

Data Types

char

subsystem - Subsystem name

string

Subsystem name specified as a string

Example: 'rtwdemo_counter/Amplifier'

Data Types

char

Name-Value Pair Arguments

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

Each `Name`, `Value` argument corresponds to a parameter on the Configuration Parameters **Code Generation > Report** pane. When the configuration parameter `GenerateReport` is on, the parameters are enabled. The `Name`, `Value` arguments are used only for generating the current report. The arguments will override, but not modify, the parameters in the model configuration. The following parameters require an Embedded Coder® license.

Example:

`'GenerateWebview','on','GenerateCodeMetricsReport','on'` includes a model Web view and static code metrics in the code generation report.

Navigation

'IncludeHyperlinkInReport' - Code-to-model hyperlinks

'off' | 'on'

Code-to-model hyperlinks, specified as 'on' or 'off'. Specify 'on' to include code-to-model hyperlinks in the code generation report. The hyperlinks link code to the corresponding blocks, Stateflow® objects, and MATLAB functions in the model diagram. For more information see “Code-to-model” on page 4-57.

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” on page 4-59.

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” on page 4-62.

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” on page 4-63.

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” on page 4-65.

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” on page 4-67.

Example: `'GenerateTraceReportsSf','on'`

Data Types

char

'GenerateTraceReportEm1' - 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” on page 4-69.

Example: `'GenerateTraceReportEm1','on'`

Data Types

char

Metrics

'GenerateCodeMetricsReport' - Static code metrics

'off' | 'on'

Static code metrics, specified as 'on' or 'off'. Specify 'on' to include static code metrics in the code generation report. For more information, see “Static code metrics” on page 4-71.

Example: 'GenerateCodeMetricsReport', 'on'

Data Types

char

Examples

Generate Code Generation Report for Model

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Build the model. The model is configured to create and open a code generation report.

```
rtwbuild('rtwdemo_counter');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report.

```
coder.report.generate('rtwdemo_counter');
```

Generate Code Generation Report for Subsystem

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

Build the subsystem. The model is configured to create and open a code generation report.

```
rtwbuild('rtwdemo_counter/Amplifier');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report for the subsystem.

```
coder.report.generate('rtwdemo_counter/Amplifier');
```

Generate Code Generation Report to Include Static Code Metrics Report

Generate a code generation report to include a static code metrics report after the build process, without modifying the model.

Open the model `rtwdemo_hyperlinks`.

```
open rtwdemo_hyperlinks
```

Build the model. The model is configured to create and open a code generation report.

```
rtwbuild('rtwdemo_hyperlinks');
```

Close the code generation report.

```
coder.report.close;
```

Generate a code generation report that includes the static code metrics report.

```
coder.report.generate('rtwdemo_hyperlinks',  
'GenerateCodeMetricsReport','on');
```

The code generation report opens. In the left navigation pane, click **Static Code Metrics Report** to view the report.

coder.report.generate

See Also

`coder.report.open` | `coder.report.close`

Concepts

- “Reports for Code Generation”
- “Generate a Code Generation Report”
- “Generate Code Generation Report After Build Process”

Purpose	Open existing HTML code generation report
Syntax	<code>coder.report.open(model)</code> <code>coder.report.open(subsystem)</code>
Description	<p><code>coder.report.open(model)</code> opens a code generation report for the <code>model</code>. The build folder for the model must be present in the current working folder.</p> <p><code>coder.report.open(subsystem)</code> opens a code generation report for the <code>subsystem</code>. The build folder for the subsystem must be present in the current working folder.</p>
Input Arguments	<p>model - Model name string Model name specified as a string Example: <code>'rtwdemo_counter'</code></p> <p>Data Types char</p> <p>subsystem - Subsystem name string Subsystem name specified as a string Example: <code>'rtwdemo_counter/Amplifier'</code></p> <p>Data Types char</p>
Examples	<p>Open code generation report for a model</p> <p>After generating code for <code>rtwdemo_counter</code>, open a code generation report for the model.</p> <pre>coder.report.open('rtwdemo_counter')</pre>

coder.report.open

Open code generation report for a subsystem

Open a code generation report for the subsystem 'Amplifier' in model 'rtwdemo_counter'.

```
coder.report.open('rtwdemo_counter/Amplifier')
```

See Also

[coder.report.close](#) | [coder.report.generate](#)

Concepts

- “Reports for Code Generation”
- “Open Code Generation Report”

Purpose Files and folders in current IDE window

Syntax `IDE_Obj.dir`
`d=IDE_Obj.dir`

IDEs This function supports the following IDEs:

Description `IDE_Obj.dir` lists the files and folders in the IDE working folder, where `IDE_Obj` is the object that references the IDE. `IDE_Obj` can be either a single object, or a vector of objects. When `IDE_Obj` is a vector, `dir` returns the files and folders referenced by each object.

`d=IDE_Obj.dir` returns the list of files and folders as an M-by-1 structure in `d` with the fields for each file and folder shown in the following table.

Field Name	Description
<code>name</code>	Name of the file or folder.
<code>date</code>	Date of most recent file or folder modification.
<code>bytes</code>	Size of the file in bytes. Folders return 0 for the number of bytes.
<code>isdirectory</code>	0 if it is a file, 1 if it is a folder.
<code>datenum</code>	Code Composer Studio IDE also returns the modification date as a MATLAB serial date number.

To view the entries in structure `d`, use an index in the syntax at the MATLAB prompt, as shown by the following examples.

- `d(3)` returns the third element in the structure.
- `d(10)` returns the tenth element in the structure `d`.
- `d(4).date` returns the `date` field value for the fourth structure element.

dir

See Also

[open](#)

Purpose Properties of IDE handle

Syntax `IDE_Obj.display()`

IDEs This function supports the following IDEs:

Description `IDE_Obj.display()` displays the properties and property values of the IDE handle `IDE_Obj`.

For example, after you creating `IDE_Obj` with a constructor, using the `display` method with `IDE_Obj` returns a set of properties and values:

```
IDE_Obj.display
```

```
IDE Object:
  Property1      : valuea
  Property2      : valueb
  Property3      : valuec
  Property4      : valued
```

See Also `get`

findBuildArg

Purpose Search for a specific build argument in model build information

Syntax `[identifier, value] = findBuildArg(buildinfo, buildArgName)`

Input Arguments

buildinfo
Build information returned by RTW.BuildInfo.

buildArgName
A character array which specifies the name of the build argument that you want to find.

Output Arguments Build argument found in the model build information. The function returns the build argument in two vectors.

Vector	Description
<i>identifier</i>	Name of the build argument that the function finds
<i>value</i>	Value of the build argument

Description The `findBuildArg` function searches for a build argument stored in the model build information. If the build argument is present in the model build information, the function returns the name and value.

See Also `getBuildArgs`

How To

- “Customize Post-Code-Generation Build Processing”

Purpose	Find and add include (header) files to build information object
Syntax	<code>findIncludeFiles(buildinfo, extPatterns)</code> <code>extPatterns</code> is optional.
Arguments	<p><i>buildinfo</i> Build information returned by <code>RTW.BuildInfo</code>.</p> <p><i>extPatterns</i> (optional) A cell array of character arrays that specify patterns of file name extensions for which the function is to search. Each pattern</p> <ul style="list-style-type: none">• Must start with <code>*</code>.• Can include a combination of alphanumeric and underscore (<code>_</code>) characters <p>The default pattern is <code>*.h</code>.</p> <p>Examples of valid patterns include</p> <pre>*.h *.hpp *.x*</pre>
Description	<p>The <code>findIncludeFiles</code> function</p> <ul style="list-style-type: none">• Searches for include files, based on specified file name extension patterns, in source and include paths recorded in the model build information object• Adds the files found, along with their full paths, to the build information object• Deletes duplicate entries
Examples	Find include files with filename extension <code>.h</code> that are in build information object <code>myModelBuildInfo</code> , and add the full paths for the files found to the object.

findIncludeFiles

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, {fullfile(pwd,...
'mycustomheaders')}, 'myheaders');
findIncludeFiles(myModelBuildInfo);
headerfiles = getIncludeFiles(myModelBuildInfo, true, false);
headerfiles
headerfiles =
    'W:\work\mycustomheaders\myheader.h'
```

See Also

[addIncludeFiles](#) | [getIncludeFiles](#) | [packNGo](#)

How To

- “Customize Post-Code-Generation Build Processing”

Purpose

Halt program execution by processor

Syntax

```
IDE_Obj.halt  
IDE_Obj.halt(timeout)
```

IDEs

This function supports the following IDEs:

Description

IDE_Obj.halt stops the program running on the processor. After you issue this command, MATLAB waits for a response from the processor that the processor has stopped. By default, the wait time is 10 seconds. If 10 seconds elapses before the response arrives, MATLAB returns an error. In this syntax, the timeout period defaults to the global timeout period specified in *IDE_Obj*. Use *IDE_Obj*.get to determine the global timeout period. However, the processor usually stops in spite of the error message.

To resume processing after you halt the processor, use *run*. Also, the *IDE_Obj*.read('pc') function can determine the memory address where the processor stopped after you use *halt*.

IDE_Obj.halt(timeout) immediately stops program execution by the processor. After the processor stops, *halt* returns to the host. *timeout* defines, in seconds, how long the host waits for the processor to stop running. If the processor does not stop within the specified timeout period, the routine returns with a timeout error.

Examples

Use one of the provided example programs to show how *halt* works. Load and run one of the example projects. At the MATLAB prompt, check whether the program is running on the processor.

```
IDE_Obj.isrunning
```

```
ans =
```

```
1
```

```
IDE_Obj.isrunning % Alternate syntax for checking the run status.
```

halt

```
ans =  
  
    1  
IDE_Obj.halt % Stop the running application on the processor.  
IDE_Obj.isrunning
```

```
ans =  
  
    0
```

Issuing the halt stops the process on the processor. Checking in the IDE confirms that the process has stopped.

See Also

`isrunning` | `run`

Purpose Insert debug point in file

Syntax `IDE_Obj.insert(addr,type,timeout)`
`IDE_Obj.insert(addr)`

IDEs This function supports the following IDEs:

Description `IDE_Obj.insert(addr,type,timeout)` places a debug point at the provided address of the processor. The `IDE_Obj` handle defines the processor that will receive the new debug point. The debug point location is defined by `addr`, the desired memory address. The IDEs support several types of debug points. Refer to your IDE help documentation for information on their respective behavior. The following table shows which debug types each IDE supports.

	CCS IDE	MULTI	VisualDSP++
'break' (default)	Yes	Yes	Yes
'watch'		Yes	
'probe'	Yes		

The `timeout` parameter defines how long to wait (in seconds) for the insert to complete. If this period is exceeded, the routine returns immediately with a timeout error. In general the action (insert) still occurs, but the timeout value gave insufficient time to verify the completion of the action.

`IDE_Obj.insert(addr)` same as the preceding example, except the `timeout` value defaults to the timeout property specified by the `IDE_Obj` object. Use `IDE_Obj.get('timeout')` to examine this default timeout value.

`IDE_Obj.insert(file,line)` same as the preceding example, except the timeout value defaults to the timeout property specified by the `IDE_Obj` object. Use `IDE_Obj.get('timeout')` to examine this default timeout value.

insert

See Also

address | run

Purpose Determine whether processor is executing process

Syntax `IDE_Obj.isrunning`

IDEs This function supports the following IDEs:

Description `IDE_Obj.isrunning` returns 1 when the processor is executing a program. When the processor is halted, `isrunning` returns 0.

Examples `isrunning` lets you determine whether the processor is running. After you load a program to the processor, use `isrunning` to verify that the program is running.

```
IDE_Obj.load('program.exe', 'program')
IDE_Obj.run
IDE_Obj.isrunning
```

```
ans =
```

```
    1
IDE_Obj.halt
IDE_Obj.isrunning
```

```
ans =
```

```
    0
```

See Also `halt` | `load` | `run`

getBuildArgs

Purpose Build arguments from model build information

Syntax `[identifiers, values] = getBuildArgs(buildinfo, includeGroupIDs, excludeGroupIDs)`
includeGroupIDs and *excludeGroupIDs* are optional.

Input Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

includeGroupIDs (optional)
A cell array which specifies group IDs of build arguments that you want the function to return.

excludeGroupIDs (optional)
A cell array which specifies group IDs of build arguments that you do not want the function to return.

Output Arguments Build arguments stored in the model build information. The function returns the build arguments in two vectors.

Vector	Description
<i>identifiers</i>	Names of the build arguments
<i>values</i>	Values of the build arguments

Description The `getBuildArgs` function returns build arguments stored in the model build information. Using optional *includeGroupIDs* and *excludeGroupIDs* arguments, you can selectively include or exclude groups from the build arguments returned by the function.

If you choose to specify *excludeGroupIDs* and omit *includeGroupIDs*, specify a null string (' ') for *includeGroupIDs*.

See Also `findBuildArg`

How To

- “Customize Post-Code-Generation Build Processing”

Purpose	Compiler options from model build information
Syntax	<pre>options = getCompileFlags(buildinfo, includeGroups, excludeGroups)</pre> <p><i>includeGroups</i> and <i>excludeGroups</i> are optional.</p>
Input Arguments	<p><i>buildinfo</i> Build information returned by RTW.BuildInfo.</p> <p><i>includeGroups</i> (optional) A character array or cell array of character arrays that specifies groups of compiler flags you want the function to return.</p> <p><i>excludeGroups</i> (optional) A character array or cell array of character arrays that specifies groups of compiler flags you do not want the function to return.</p>
Output Arguments	Compiler options stored in the model build information.
Description	<p>The <code>getCompileFlags</code> function returns compiler options stored in the model build information. Using optional <i>includeGroups</i> and <i>excludeGroups</i> arguments, you can selectively include or exclude groups of options the function returns.</p> <p>If you choose to specify <i>excludeGroups</i> and omit <i>includeGroups</i>, specify a null string ('') for <i>includeGroups</i>.</p>
Examples	<ul style="list-style-type: none">• Get the compiler options stored in build information <code>myModelBuildInfo</code>. <pre>myModelBuildInfo = RTW.BuildInfo; addCompileFlags(myModelBuildInfo, {'-Zi -Wall' '-03'}, ... 'OPTS'); compflags=getCompileFlags(myModelBuildInfo); compflags</pre>

getCompileFlags

```
compflags =  
    '-Zi -Wall' '-03'
```

- Get the compiler options stored with the group name Debug in build information myModelBuildInfo.

```
myModelBuildInfo = RTW.BuildInfo;  
addCompileFlags(myModelBuildInfo, {'-Zi -Wall' '-03'}, ...  
    {'Debug' 'MemOpt'});  
compflags=getCompileFlags(myModelBuildInfo, 'Debug');  
compflags
```

```
compflags =  
    '-Zi -Wall'
```

- Get the compiler options stored in build information myModelBuildInfo, except those with the group name Debug.

```
myModelBuildInfo = RTW.BuildInfo;  
addCompileFlags(myModelBuildInfo, {'-Zi -Wall' '-03'}, ...  
    {'Debug' 'MemOpt'});  
compflags=getCompileFlags(myModelBuildInfo, '', 'Debug');  
compflags
```

```
compflags =  
    '-03'
```

See Also

[addCompileFlags](#) | [getDefines](#) | [getLinkFlags](#)

How To

- “Customize Post-Code-Generation Build Processing”

Purpose Preprocessor macro definitions from model build information

Syntax `[macrodefs, identifiers, values] = getDefines(buildinfo, includeGroups, excludeGroups)`
includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
 Build information returned by RTW.BuildInfo.

includeGroups (optional)
 A character array or cell array of character arrays that specifies groups of macro definitions you want the function to return.

excludeGroups (optional)
 A character array or cell array of character arrays that specifies groups of macro definitions you do not want the function to return.

Output Arguments Preprocessor macro definitions stored in the model build information. The function returns the macro definitions in three vectors.

Vector	Description
<i>macrodefs</i>	Complete macro definitions with -D prefix
<i>identifiers</i>	Names of the macros
<i>values</i>	Values assigned to the macros (anything specified to the right of the first equals sign) ; the default is an empty string (' ')

getDefines

Description

The `getDefines` function returns preprocessor macro definitions stored in the model build information. When the function returns a definition, it automatically

- Prepends a `-D` to the definition if the `-D` was not specified when the definition was added to the build information
- Changes a lowercase `-d` to `-D`

Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of definitions the function is to return.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string (`' '`) for *includeGroups*.

Examples

- Get the preprocessor macro definitions stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
    {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, 'OPTS');
[defs names values]=getDefines(myModelBuildInfo);
defs
    '-DPROTO=first' '-DDEBUG' '-Dtest' '-DPRODUCTION'

names
    'PROTO'
    'DEBUG'
    'test'
    'PRODUCTION'

values
```

```
values =
    'first'
    ''
    ''
    ''
```

- Get the preprocessor macro definitions stored with the group name Debug in build information myModelBuildInfo.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
    {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
    {'Debug' 'Debug' 'Debug' 'Release'});
[defs names values]=getDefines(myModelBuildInfo, 'Debug');
defs
```

```
defs =
    '-DPROTO=first'    '-DDEBUG'    '-Dtest'
```

- Get the preprocessor macro definitions stored in build information myModelBuildInfo, except those with the group name Debug.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, ...
    {'PROTO=first' '-DDEBUG' 'test' '-dPRODUCTION'}, ...
    {'Debug' 'Debug' 'Debug' 'Release'});
[defs names values]=getDefines(myModelBuildInfo, '', 'Debug');
defs
```

```
defs =
    '-DPRODUCTION'
```

See Also

[addDefines](#) | [getCompileFlags](#) | [getLinkFlags](#)

getDefines

How To

- “Customize Post-Code-Generation Build Processing”

Purpose List of files from model build information

Syntax `[fPathNames, names] = getFullFileList(buildinfo, fcase)`
fcase is optional.

Input Arguments

buildinfo
 Build information returned by RTW.BuildInfo.

fcase (optional)
 The string 'source', 'include', or 'nonbuild'. If the argument is omitted, the function returns files from the model build information.

If You Specify...	The Function...
'source'	Returns source files from the model build information.
'include'	Returns include files from the model build information.
'nonbuild'	Returns nonbuild files from the model build information.

Output Arguments Fully-qualified file paths and file names for files stored in the model build information.

Note It is not required to resolve the path of every file in the model build information, because the makefile for the model build will resolve file locations based on source paths and rules. Therefore, `getFullFileList` returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

getFullFileList

Description

The `getFullFileList` function returns the fully-qualified paths and names of all files, or files of a selected type (source, include, or nonbuild), stored in the model build information.

The `packNGo` function calls `getFullFileList` to return a list of files in the model build information before processing files for packaging.

Examples

List the files stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;  
[fPathNames, names] = getFullFileList(myModelBuildInfo);
```

How To

- “Customize Post-Code-Generation Build Processing”

Purpose Include files from model build information

Syntax `files = getIncludeFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)`
includeGroups and *excludeGroups* are optional.

Input Arguments *buildinfo*
Build information returned by RTW.BuildInfo.

concatenatePaths
The logical value true or false.

If You Specify...	The Function...
true	Concatenates and returns each filename with its corresponding path.
false	Returns only filenames.

Note It is not required to resolve the path of every file in the model build information, because the makefile for the model build will resolve file locations based on source paths and rules. Therefore, specifying true for *concatenatePaths* returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getIncludeFiles`.

replaceMatlabroot
The logical value true or false.

getIncludeFiles

If You Specify...	The Function...
true	Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation folder.
false	Does not replace the token <code>\$(MATLAB_ROOT)</code> .

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of include files you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of include files you do not want the function to return.

Output Arguments

Names of include files stored in the model build information. The names include files you added using the `addIncludeFiles` function and, if you called the `packNGo` function, files `packNGo` found and added while packaging model code.

Description

The `getIncludeFiles` function returns the names of include files stored in the model build information. Use the *concatenatePaths* and *replaceMatlabroot* arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of include files the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string (`' '`) for *includeGroups*.

Examples

- Get the include paths and filenames stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, {'etc.h' 'etc_private.h'...
'mytypes.h'}, {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
```

```
    '/common/lib'}, {'etc' 'etc' 'shared'}));  
incfiles=getIncludeFiles(myModelBuildInfo, true, false);  
incfiles  
  
incfiles =  
  
    [1x22 char]    [1x36 char]    [1x21 char]
```

- Get the names of include files in group etc that are stored in build information myModelBuildInfo.

```
myModelBuildInfo = RTW.BuildInfo;  
addIncludeFiles(myModelBuildInfo, {'etc.h' 'etc_private.h'...  
    'mytypes.h'}, {'/etc/proj/etclib' '/etcproj/etc/etc_build'...  
    '/common/lib'}, {'etc' 'etc' 'shared'});  
incfiles=getIncludeFiles(myModelBuildInfo, false, false,...  
    'etc');  
incfiles  
  
incfiles =  
  
    'etc.h'    'etc_private.h'
```

See Also

[addIncludeFiles](#) | [findIncludeFiles](#) | [getIncludePaths](#) | [getSourceFiles](#) | [getSourcePaths](#) | [updateFilePathsAndExtensions](#)

How To

- “Customize Post-Code-Generation Build Processing”

getIncludePaths

Purpose Include paths from model build information

Syntax `files=getIncludePaths(buildinfo, replaceMatlabroot, includeGroups, excludeGroups)`
includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
Build information returned by RTW.BuildInfo.

replaceMatlabroot
The logical value true or false.

If You Specify...	The Function...
true	Replaces the token \$(MATLAB_ROOT) with the absolute path string for your MATLAB installation folder.
false	Does not replace the token \$(MATLAB_ROOT).

includeGroups (optional)
A character array or cell array of character arrays that specifies groups of include paths you want the function to return.

excludeGroups (optional)
A character array or cell array of character arrays that specifies groups of include paths you do not want the function to return.

Output Arguments Paths of include files stored in the model build information.

Description The `getIncludePaths` function returns the names of include file paths stored in the model build information. Use the *replaceMatlabroot* argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of include file paths the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string (' ') for *includeGroups*.

Examples

- Get the include paths stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo, {'/etc/proj/etcclib'...
    '/etcproj/etc/etc_build' '/common/lib'},...
    {'etc' 'etc' 'shared'});
incpaths=getIncludePaths(myModelBuildInfo, false);
incpaths
```

```
incpaths =
```

```
    '\etc\proj\etcclib'    [1x22 char]    '\common\lib'
```

- Get the paths in group `shared` that are stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo, {'/etc/proj/etcclib'...
    '/etcproj/etc/etc_build' '/common/lib'},...
    {'etc' 'etc' 'shared'});
incpaths=getIncludePaths(myModelBuildInfo, false, 'shared');
incpaths
```

```
incpaths =
```

```
    '\common\lib''
```

See Also

[addIncludePaths](#) | [getIncludeFiles](#) | [getSourceFiles](#) | [getSourcePaths](#)

How To

- “Customize Post-Code-Generation Build Processing”

getLinkFlags

Purpose	Link options from model build information
Syntax	<pre>options=getLinkFlags(buildinfo, includeGroups, excludeGroups)</pre> <p><i>includeGroups</i> and <i>excludeGroups</i> are optional.</p>
Input Arguments	<p><i>buildinfo</i> Build information returned by RTW.BuildInfo.</p> <p><i>includeGroups</i> (optional) A character array or cell array that specifies groups of linker flags you want the function to return.</p> <p><i>excludeGroups</i> (optional) A character array or cell array that specifies groups of linker flags you do not want the function to return. To exclude groups and not include specific groups, specify an empty cell array ('') for <i>includeGroups</i>.</p>
Output Arguments	Linker options stored in the model build information.
Description	<p>The <code>getLinkFlags</code> function returns linker options stored in the model build information. Using optional <i>includeGroups</i> and <i>excludeGroups</i> arguments, you can selectively include or exclude groups of options the function returns.</p> <p>If you choose to specify <i>excludeGroups</i> and omit <i>includeGroups</i>, specify a null string ('') for <i>includeGroups</i>.</p>

Examples

- Get the linker options stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, {'-MD -Gy' '-T'}, 'OPTS');
linkflags=getLinkFlags(myModelBuildInfo);
linkflags
```

```
linkflags =
```

```
    '-MD -Gy'    '-T'
```

- Get the linker options stored with the group name `Debug` in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, {'-MD -Gy' '-T'}, ...
    {'Debug' 'MemOpt'});
linkflags=getLinkFlags(myModelBuildInfo, {'Debug'});
linkflags
```

```
linkflags =
```

```
    '-MD -Gy'
```

- Get the linker options stored in build information `myModelBuildInfo`, except those with the group name `Debug`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, {'-MD -Gy' '-T'}, ...
    {'Debug' 'MemOpt'});
linkflags=getLinkFlags(myModelBuildInfo, '', {'Debug'});
linkflags
```

```
linkflags =
```

```
    '-T'
```

See Also

[addLinkFlags](#) | [getCompileFlags](#) | [getDefines](#)

getLinkFlags

How To

- “Customize Post-Code-Generation Build Processing”

Purpose Nonbuild-related files from model build information

Syntax `files=getNonBuildFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)`
includeGroups and *excludeGroups* are optional.

Input Arguments *buildinfo*
Build information returned by RTW.BuildInfo.

concatenatePaths
The logical value true or false.

If You Specify...	The Function...
true	Concatenates and returns each filename with its corresponding path.
false	Returns only filenames.

Note It is not required to resolve the path of every file in the model build information, because the makefile for the model build will resolve file locations based on source paths and rules. Therefore, specifying true for *concatenatePaths* returns the path for each file only if a path was explicitly associated with the file when it was added.

replaceMatlabroot
The logical value true or false.

getNonBuildFiles

If You Specify...	The Function...
true	Replaces the token \$(MATLAB_ROOT) with the absolute path string for your MATLAB installation folder.
false	Does not replace the token \$(MATLAB_ROOT).

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of nonbuild files you do not want the function to return.

Output Arguments

Names of nonbuild files stored in the model build information.

Description

The `getNonBuildFiles` function returns the names of nonbuild-related files, such as DLL files required for a final executable, or a README file, stored in the model build information. Use the *concatenatePaths* and *replaceMatlabroot* arguments to control whether the function includes paths and your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of nonbuild files the function returns.

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string ('') for *includeGroups*.

Examples

Get the nonbuild filenames stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;  
addNonBuildFiles(myModelBuildInfo, {'readme.txt' 'myutility1.dll'...  
'myutility2.dll'});
```

```
nonbuildfiles=getNonBuildFiles(myModelBuildInfo, false, false);  
nonbuildfiles
```

```
nonbuildfiles =
```

```
    'readme.txt'    'myutility1.dll'    'myutility2.dll'
```

See Also

`addNonBuildFiles`

How To

- “Customize Post-Code-Generation Build Processing”

getSourceFiles

Purpose Source files from model build information

Syntax `srcfiles=getSourceFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, excludeGroups)`
includeGroups and *excludeGroups* are optional.

Input Arguments *buildinfo*
Build information returned by RTW.BuildInfo.

concatenatePaths
The logical value true or false.

If You Specify...	The Function...
true	Concatenates and returns each filename with its corresponding path.
false	Returns only filenames.

Note It is not required to resolve the path of every file in the model build information, because the makefile for the model build will resolve file locations based on source paths and rules. Therefore, specifying true for *concatenatePaths* returns the path for each file only if a path was explicitly associated with the file when it was added, or if you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getSourceFiles`.

replaceMatlabroot
The logical value true or false.

If You Specify...	The Function...
true	Replaces path tokens, such as \$(MATLAB_ROOT) and \$(START_DIR), with the absolute path string.
false	Does not replace path tokens with the absolute path string.

includeGroups (optional)

A character array or cell array of character arrays that specifies groups of source files you want the function to return.

excludeGroups (optional)

A character array or cell array of character arrays that specifies groups of source files you do not want the function to return.

Output Arguments

Names of source files stored in the model build information.

Description

The `getSourceFiles` function returns the names of source files stored in the model build information. Use the `concatenatePaths` and `replaceMatlabroot` arguments to control whether the function includes paths and expansions of path tokens in the output it returns. Using optional `includeGroups` and `excludeGroups` arguments, you can selectively include or exclude groups of source files the function returns.

If you choose to specify `excludeGroups` and omit `includeGroups`, specify a null string ('') for `includeGroups`.

Examples

- Get the source paths and filenames stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo,...
{'test1.c' 'test2.c' 'driver.c'}, '',...
{'Tests' 'Tests' 'Drivers'});
srcfiles=getSourceFiles(myModelBuildInfo, false, false);
```

getSourceFiles

```
srcfiles  
  
srcfiles =  
  
    'test1.c'    'test2.c'    'driver.c'
```

- Get the names of source files in group tests that are stored in build information myModelBuildInfo.

```
myModelBuildInfo = RTW.BuildInfo;  
addSourceFiles(myModelBuildInfo, {'test1.c' 'test2.c'...  
    'driver.c'}, {'/proj/test1' '/proj/test2'...  
    '/drivers/src'}, {'tests', 'tests', 'drivers'});  
incfiles=getSourceFiles(myModelBuildInfo, false, false,...  
    'tests');  
incfiles  
  
incfiles =  
  
    'test1.c'    'test2.c'
```

See Also

[addSourceFiles](#) | [getIncludeFiles](#) | [getIncludePaths](#) |
[getSourcePaths](#) | [updateFilePathsAndExtensions](#)

How To

- “Customize Post-Code-Generation Build Processing”

Purpose Source paths from model build information

Syntax `files=getSourcePaths(buildinfo, replaceMatlabroot, includeGroups, excludeGroups)`
includeGroups and *excludeGroups* are optional.

Input Arguments *buildinfo*
 Build information returned by RTW.BuildInfo.

replaceMatlabroot
 The logical value true or false.

If You Specify...	The Function...
true	Replaces the token \$(MATLAB_ROOT) with the absolute path string for your MATLAB installation folder.
false	Does not replace the token \$(MATLAB_ROOT).

includeGroups (optional)
 A character array or cell array of character arrays that specifies groups of source paths you want the function to return.

excludeGroups (optional)
 A character array or cell array of character arrays that specifies groups of source paths you do not want the function to return.

Output Arguments Paths of source files stored in the model build information.

Description The getSourcePaths function returns the names of source file paths stored in the model build information. Use the *replaceMatlabroot* argument to control whether the function includes your MATLAB root definition in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of source file paths the function returns.

getSourcePaths

If you choose to specify *excludeGroups* and omit *includeGroups*, specify a null string ('') for *includeGroups*.

Examples

- Get the source paths stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, {'/proj/test1'...
'/proj/test2' '/drivers/src'}, {'tests' 'tests'...
'drivers'});
srcpaths=getSourcePaths(myModelBuildInfo, false);
srcpaths

srcpaths =

    '\proj\test1'    '\proj\test2'    '\drivers\src'
```

- Get the paths in group `tests` that are stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, {'/proj/test1'...
'/proj/test2' '/drivers/src'}, {'tests' 'tests'...
'drivers'});
srcpaths=getSourcePaths(myModelBuildInfo, true, 'tests');
srcpaths

srcpaths =

    '\proj\test1'    '\proj\test2'
```

- Get a path stored in build information `myModelBuildInfo`. First get the path without replacing `$(MATLAB_ROOT)` with an absolute path, then get it with replacement. The MATLAB root folder in this case is `\\myserver\myworkspace\matlab`.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, fullfile(matlabroot,...
'rtw', 'c', 'src'));
```

```
srcpaths=getSourcePaths(myModelBuildInfo, false);
srcpaths{:}

ans =

$(MATLAB_ROOT)\rtw\c\src

srcpaths=getSourcePaths(myModelBuildInfo, true);
srcpaths{:}

ans =

\\myserver\myworkspace\matlab\rtw\c\src
```

See Also

[addSourcePaths](#) | [getIncludeFiles](#) | [getIncludePaths](#) | [getSourceFiles](#)

How To

- “Customize Post-Code-Generation Build Processing”

load

Purpose	Load program file onto processor
Syntax	<code>IDE_Obj.load(filename,timeout)</code>
IDEs	This function supports the following IDEs:
Description	<p><code>IDE_Obj.load(filename,timeout)</code> loads the file specified by the <i>filename</i> argument to the processor.</p> <p>The <i>filename</i> argument can include a full path to the file, or the name of a file in the IDE working folder.</p> <p>With the VisualDSP++, MULTI, and Code Composer Studio IDEs, you can use the <code>cd</code> method to check or modify the IDE working folder.</p> <p>For MULTI, you can add an <i>option</i> argument after <i>filename</i> to specify options for the 'prepare_target' command in MULTI debugger. Refer to the MULTI documentation for information on 'prepare_target'.</p> <p>Only use <code>load</code> with program files created by the IDE build process.</p> <p>The <i>timeout</i> argument defines the number of seconds MATLAB waits for the load process to complete. If the time-out period expires before the load process returns a completion message, MATLAB generates an error and returns. Usually the program load process works in spite of the error message.</p> <p>If you omit the <i>timeout</i> argument, <code>load</code> uses the <code>timeout</code> property of the IDE handle object, which you can get by entering <code>IDE_Obj.get('timeout')</code>.</p>
Examples	<pre>IDE_Obj.load(programfile) run(id)</pre>
See Also	<code>dir</code> <code>open</code>

Purpose

Initialization entry point in generated code for Simulink model

Syntax

```
void model_initialize(void)
```

Calling Interfaces

The calling interface generated for this function differs depending on the value of the model parameter **Code interface packaging**:

- **C++ class** (default for C++ language) — Generated function is encapsulated into a C++ class method. Required model data is encapsulated into C++ class attributes.
- **Nonreusable function** (default for C language) — Generated function passes (void). Model data structures are statically allocated, global, and accessed directly in the model code.
- **Reusable function** — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the *model.h* header file.

For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the function. They can be included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.

For a GRT-based model, the generated *model.c* source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, you can use the **Generate function to allocate model data** option to control whether an allocation function is generated.

Note If you have an Embedded Coder license, for Nonreusable function code interface packaging, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box. For more information, see “Function Prototype Control” in the Embedded Coder documentation.

model_initialize

Description The generated *model_initialize* function contains the model initialization code for a Simulink model and should be called once at the beginning of model execution.

See Also `model_step` | `model_terminate`

How To • “Entry-Point Functions and Scheduling”

Purpose Step routine entry point in generated code for Simulink model

Syntax

```
void model_step(void)
void model_stepN(void)
```

Calling Interfaces The *model_step* default function prototype varies depending on the **Tasking mode for periodic sample times** (SolverMode) parameter specified for the model:

Tasking Mode	Function Prototype
SingleTasking (single-rate or multirate)	void <i>model_step</i> (void);
MultiTasking (multirate)	void <i>model_stepN</i> (void); (<i>N</i> is a task identifier)

The calling interface generated for this function also differs depending on the value of the model parameter **Code interface packaging**:

- **C++ class** (default for C++ language) — Generated function is encapsulated into a C++ class method. Required model data is encapsulated into C++ class attributes.
- **Nonreusable function** (default for C language) — Generated function passes (void). Model data structures are statically allocated, global, and accessed directly in the model code.
- **Reusable function** — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the *model.h* header file.

For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the function. They can be included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.

Note If you have an Embedded Coder license:

- For Nonreusable function code interface packaging, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box. For more information, see “Function Prototype Control” in the Embedded Coder documentation.
 - For C++ class code interface packaging, you can use the **Configure C++ Class Interface** button and related controls on the **Interface** pane of the Configuration Parameters dialog box. For more information, see “C++ Class Interface Control” in the Embedded Coder documentation.
-

Description

The generated *model_step* function contains the output and update code for the blocks in a Simulink model. The *model_step* function computes the current value of the blocks. If logging is enabled, *model_step* updates logging variables. If the model’s stop time is finite, *model_step* signals the end of execution when the current time equals the stop time.

Under the following conditions, *model_step* does not check the current time against the stop time:

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

Therefore, if one or more of these conditions are true, the program runs indefinitely.

For an ERT-based model, the software generates a *model_step* function when the **Single output/update function** configuration option is selected (the default) in the Configuration Parameters dialog box.

model_step is designed to be called at interrupt level from *rt_OneStep*, which is assumed to be invoked as a timer ISR. *rt_OneStep* calls *model_step* to execute processing for one clock period of the model. See “*rt_OneStep* and Scheduling Considerations” in the Embedded

Coder documentation for a description of how calls to *model_step* are generated and scheduled.

Note For an ERT-based model, if the **Single output/update function** configuration option is not selected, the Embedded Coder software generates the following model entry point functions in place of *model_step*:

- *model_output*: Contains the output code for the blocks in the model
 - *model_update*: Contains the update code for the blocks in the model
-

See Also

[model_initialize](#) | [model_terminate](#)

How To

- “Entry-Point Functions and Scheduling”

model_terminate

Purpose	Termination entry point in generated code for Simulink model
Syntax	<code>void model_terminate(void)</code>
Calling Interfaces	<p>The calling interface generated for this function also differs depending on the value of the model parameter Code interface packaging:</p> <ul style="list-style-type: none">• C++ class (default for C++ language) — Generated function is encapsulated into a C++ class method. Required model data is encapsulated into C++ class attributes.• Nonreusable function (default for C language) — Generated function passes (void). Model data structures are statically allocated, global, and accessed directly in the model code.• Reusable function — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the <code>model.h</code> header file. <p>For an ERT-based model, you can use the Pass root-level I/O as parameter to control how root-level input and output arguments are passed to the function. They can be included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.</p>

Description

The generated `model_terminate` function contains the termination code for a Simulink model and should be called as part of system shutdown.

When `model_terminate` is called, blocks that have a terminate function execute their terminate code. If logging is enabled, `model_terminate` ends data logging.

The `model_terminate` function should be called only once.

For an ERT-based model, the Embedded Coder software generates the `model_terminate` function for a model when the **Terminate function required** configuration option is selected (the default) in the Configuration Parameters dialog box. If your application runs indefinitely, you do not need the `model_terminate` function. To

suppress the function, clear the **Terminate function required** configuration option in the Configuration Parameters dialog box.

See Also

`model_initialize` | `model_step`

How To

- “Entry-Point Functions and Scheduling”

new

Purpose Create project, library, or build configuration in IDE

Syntax `IDE_Obj.new('name', 'type')`

IDEs This function supports the following IDEs:

Description `IDE_Obj.new('name', 'type')` creates a project, library, or build configuration in the IDE.

The *name* argument specifies the name of the new project, library, or build configuration

The *type* argument specifies whether to create a project, library, or build configuration. The options are:

- 'project' — Executable project. Sometimes this file is called a “DSP executable file”.
- 'projlib' — Library project.
- 'projext' — External make project. Only the CCS IDE supports this option.
- 'buildcfg' — Build configuration in the active project. Only the VisualDSP++ and CCS IDEs support this option.

When *type* is 'project' or 'projlib', *name* can include the full path to the new file. You can use the path to differentiate two files with the same name. If you omit the path, the new method creates the file or project in the current IDE working folder.

If you omit the *type* argument, and the *name* argument does not include a file extension, *type* defaults to 'project'.

When *type* is 'buildcfg', use a unique name to differentiate the build configuration from other build configurations in the active project.

The new method does not support 'text' as a *type* argument.

Examples

```
IDE_Obj.new('my_project','project') #Create an IDE project, 'my_project.gpj'  
IDE_Obj.new('my_build_config','buildcfg') #Create a build configuration.
```

See Also

activate | close

open

Purpose Open project in IDE

Syntax `IDE_Obj.open(filename, filetype, timeout)`
`IDE_Obj.open(myproject)`

IDEs This function supports the following IDEs:

Description `IDE_Obj.open(filename, filetype, timeout)` opens a project in the IDE.

Use the *filename* argument to specify the file name, including the file name extension. If the *filename* does not include a file name extension, you can specify the file type using the *filetype* argument. If the file does not exist in the current project or folder path, MATLAB returns a warning and returns control to MATLAB.

For the optional *filetype* argument, you can specify the following types.

	CCS IDE	MULTI IDE	VisualDSP++ IDE
'project' — Project files	Yes	Yes	Yes
'ProjectGroup' — Project group files	No	No	Yes
'program' — Target program file (executable)	No. Use load instead.	Yes	No

If you omit the *filetype* argument, *filetype* defaults to 'project'.

The optional *timeout* argument determines the number of seconds MATLAB waits for the IDE to finish opening the file before returning an error. If you omit the *timeout* argument, the open method uses the timeout property of the IDE handle object (IDE_Obj) instead. The timeout error does not terminate the loading process on the IDE.

Note The open method does not support the 'text', 'program', or 'workspace' arguments.

Examples

IDE_Obj.open(myproject) opens the myproject project in the IDE.
dir | load | new

packNGo

Purpose Package model code in zip file for relocation

Syntax `packNGo(buildinfo, propVals...)`
propVals is optional.

Arguments *buildinfo*
Build information returned by `RTW.BuildInfo`.
propVals (optional)
A cell array of property-value pairs that specify packaging details.

To...	Specify Property...	With Value...
Package model code files in a zip file as a single, flat folder	'packType'	'flat' (default)
Package model code files hierarchically in a primary zip file that contains three secondary zip files: <ul style="list-style-type: none">• <code>m1rFiles.zip</code> — files in your <i>matlabroot</i> folder tree• <code>sDirFiles.zip</code> — files in and under your build folder• <code>otherFiles.zip</code> — required files not in the <i>matlabroot</i> or <i>start</i> folder trees	'packType'	'hierarchical' Paths for files in the secondary zip files are relative to the root folder of the primary zip file.
Specify a file name for the primary zip file	'fileName'	'string' Default: ' <i>model.zip</i> ' If you omit the <i>.zip</i> file extension, the function adds it for you.
Include only the minimal header files required to build the code in the zip file	'minimalHeaders'	true (default)
Include header files found on the include path in the zip file	'minimalHeaders'	false

To...	Specify Property...	With Value...
Direct packNGo not to error out on parse errors	'ignoreParseError'	true (default is false)
Direct packNGo not to error out if files are missing	'ignoreFileMissing'	true (default is false)

Description

The packNGo function packages the following code files in a compressed zip file so you can relocate, unpack, and rebuild them in another development environment:

- Source files (for example, .c and .cpp files)
- Header files (for example, .h and .hpp files)
- Nonbuild-related files (for example, .dll files required for a final executable and .txt informational files)
- MAT-file that contains the model build information object (.mat file)

You might use this function to relocate files so they can be recompiled for a specific target environment or rebuilt 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 named *model.zip*. You can tailor the output by specifying property name and value pairs as explained above.

After relocating the zip file, use a standard zip utility to unpack the compressed file.

Note The packNGo function potentially can modify the build information passed in the first packNGo argument. As part of packaging model code, packNGo might find additional files from source and include paths recorded in the model's build information and add them to the build information.

Examples

- Package the code files for model zingbit in the file zingbit.zip as a flat folder structure.

```
set_param('zingbit','PostCodeGenCommand','packNGo(buildInfo);');
```

Then, rebuild the model.

- Package the code files for model zingbit in the file portzingbit.zip and maintain the relative file hierarchy.

```
cd zingbit_grt_rtw;  
load buildInfo.mat  
packNGo(buildInfo, {'packType', 'hierarchical', ...  
  'fileName', 'portzingbit'});
```

Alternatives

You can configure model code packaging by selecting the **Package code and artifacts** option on the **Code Generation** pane of the Configuration Parameters dialog box.

How To

- “Customize Post-Code-Generation Build Processing”
- “Relocate Code to Another Development Environment”
- “packNGo Function Limitations”

Purpose

Read data from processor memory

Syntax

```
mem=IDE_Obj.read(address)
mem=IDE_Obj.read(...,datatype)
mem=IDE_Obj.read(...,count)
mem=IDE_Obj.read(...,memorytype)
mem=IDE_Obj.read(...,timeout)
```

IDEs

This function supports the following IDEs:

Description

`mem=IDE_Obj.read(address)` returns a block of data values from the memory space of the processor referenced by `IDE_Obj`. The block to read begins from the DSP memory location given by the *address* argument. The data is read starting from *address* without regard to type-alignment boundaries in the processor. Conversely, the byte ordering defined by the data type is automatically applied.

The *address* argument is a decimal or hexadecimal representation of a memory address in the processor. The full memory address consist of two parts:

- The start address
- The memory type

You can define the memory type value can be explicitly using a numeric vector representation of the address.

Alternatively, the `IDE_Obj` object has a default memory type value that is applied if the memory type value is not explicitly incorporated in the passed address parameter. In DSP processors with only a single memory type, it is possible to specify addresses using the abbreviated (implied memory type) format by setting the `IDE_Obj` object memory type value to zero.

Note You cannot read data from processor memory while the processor is running.

read

Provide the *address* argument either as a numerical value that is a decimal representation of the DSP memory address, or as a string that `read` converts to the decimal representation of the start address. (Refer to function `hex2dec` in the *MATLAB Function Reference*. `read` uses `hex2dec` to convert the hexadecimal string to a decimal value).

The examples in the following table show how `read` uses the *address* parameter.

address Parameter Value	Description
131082	Decimal address specification. The memory start address is 131082 and memory type is 0. This action is the same as specifying [131082 0].
[131082 1]	Decimal address specification. The memory start address is 131082 and memory type is 1.
'2000A'	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify *address* as a cell array. You can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1}=131072;  
myaddress1{2}='Program(PM) Memory';
```

```
myaddress2 myaddress2{1}='20000';  
myaddress2{2}='Program(PM) Memory';
```

```
myaddress3 myaddress3{1}=131072; myaddress3{2}=0;
```

`mem=IDE_Obj.read(...,datatype)` where the input argument `datatype` defines the interpretation of the raw values read from DSP memory. Parameter `datatype` specifies the data format of the raw memory

image. The data is read starting from `address` without regard to data type alignment boundaries in the processor. The byte ordering defined by the data type is automatically applied. This syntax supports the following MATLAB data types.

MATLAB Data Type	Description
<code>double</code>	IEEE double-precision floating point value
<code>single</code>	IEEE single-precision floating point value
<code>uint8</code>	8-bit unsigned binary integer value
<code>uint16</code>	16-bit unsigned binary integer value
<code>uint32</code>	32-bit unsigned binary integer value
<code>int8</code>	8-bit signed two's complement integer value
<code>int16</code>	16-bit signed two's complement integer value
<code>int32</code>	32-bit signed two's complement integer value

The `read` method does not coerce data type alignment. Some combinations of `address` and `datatype` will be difficult for the processor to use.

`mem=IDE_Obj.read(...,count)` adds the `count` input parameter that defines the dimensions of the returned data block `mem`. To read a block of multiple data values. Specify `count` to determine how many values to read from `address`. `count` can be a scalar value that causes `read` to return a column vector that has `count` values. You can perform multidimensional reads by passing a vector for `count`. The elements in the input vector of `count` define the dimensions of the returned data

matrix. The memory is read in column-major order. `count` defines the dimensions of the returned data array `mem` as shown in the following table.

- `n` — Read `n` values into a column vector.
- `[m,n]` — Read `m`-by-`n` values into `m` by `n` matrix in column-major order.
- `[m,n,...]` — Read a multidimensional matrix `m`-by-`n`-by...of values into an `m`-by-`n`-by...array.

To read a block of multiple data values, specify the input argument `count` that determines how many values to read from address.

`mem=IDE_Obj.read(...,memorytype)` adds an optional input argument `memorytype`. Object `IDE_Obj` has a default memory type value 0 that read applies if the memory type value is not explicitly incorporated into the passed address parameter.

In processors with only a single memory type, it is possible to specify addresses using the implied memory type format by setting the `IDE_Objmemorytype` property value to zero.

Using read with MULTI

Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
'program(pm) memory'	0	Blackfin and SHARC
'data(dm) memory'	1	SHARC
'data(dm) short word memory'	2	SHARC

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
'external data(dm) byte memory'	3	SHARC
'boot(prom) memory'	4	SHARC

`mem=IDE_Obj.read(...,timeout)` adds the optional parameter *timeout* that defines how long, in seconds, MATLAB waits for the specified read process to complete. If the time-out period expires before the read process returns a completion message, MATLAB returns an error and returns. Usually the read process works in spite of the error message.

Examples

This example reads one 16-bit integer from memory on the processor.

```
m1var = IDE_Obj.read(131072,'int16')
```

131072 is the decimal address of the data to read.

You can read more than one value at a time. This read command returns 100 32-bit integers from the address 0x20000 and plots the result in MATLAB.

```
data = IDE_Obj.read('20000','int32',100)
plot(double(data))
```

See Also

`write`

reload

Purpose Reload most recent program file to processor signal processor

Syntax
`s = IDE_Obj.reload(timeout)`
`s = IDE_Obj.reload`

IDEs This function supports the following IDEs:

Description `s = IDE_Obj.reload(timeout)` resends the most recently loaded program file to the processor. If you have not loaded a program file in the current session (so there is no previously loaded file), `reload` returns the null entry [] in `s` indicating that it could not load a file to the processor. Otherwise, `s` contains the full path name to the program file. After you reset your processor or after an event produces changes in your processor memory, use `reload` to restore the program file to the processor for execution.

To limit the time the IDE spends trying to reload the program file to the processor, `timeout` specifies how long the load process can take. If the load process exceeds the timeout limit, the IDE stops trying to load the program file and returns an error stating that the time period expired. Exceeding the allotted time for the reload operation usually indicates that the reload was complete but the IDE did not receive confirmation before the timeout period passed.

`s = IDE_Obj.reload` reloads the most recent program file, using the `timeout` value set when you created link `IDE_Obj`, the global timeout setting.

Using reload with Multiprocessor Boards

When your board contains more than one processor, `reload` calls the reloading function for each processor represented by `IDE_Obj`, reloading the most recently loaded program on each processor.

This action is the same as calling `reload` for each processor individually through IDE handle objects for each one.

Examples After you create an object that connects to the IDE, use the available methods to reload your most recently loaded project. If you have not

loaded a project in this session, `reload` returns an error and an empty value for `s`. Loading a project eliminates the error. First, create an IDE handle object, such as `IDE_Obj`, using the constructor for your IDE.

```
s=IDE_Obj.reload(23)
Warning: No action taken - load a valid Program file before
you reload...

s =

    ''

IDE_Obj.open('D:\ti\tutorial\sim62xx\gelsolid\hellodsp.pjt','project')

IDE_Obj.build

IDE_Obj.load('hellodsp.pjt') #This file extension varies by IDE
IDE_Obj.halt
s=IDE_Obj.reload(23)

s =

D:\ti\tutorial\sim62xx\gelsolid\Debug\hellodsp.out

load | open
```

remove

Purpose	Remove file, project, or breakpoint
Syntax	<pre>IDE_Obj.remove(filename,filetype) IDE_Obj.remove(addr,debugtype,timeout) IDE_Obj.remove(filename,line,debugtype,timeout) IDE_Obj.remove(all,break)</pre>
IDEs	This function supports the following IDEs:
Description	<p><i>IDE_Obj.remove(filename,filetype)</i> deletes a file from the active project in the IDE or deletes the project.</p> <p><i>IDE_Obj.remove(addr,debugtype,timeout)</i> removes a debug point from an address in the program.</p> <p><i>IDE_Obj.remove(filename,line,debugtype,timeout)</i> removes a debug point from a line in a source file.</p> <p><i>IDE_Obj.remove(all,break)</i> removes the breakpoints and waits for completion.</p>
Input Arguments	<p>IDE_Obj</p> <p>Enter the name of the IDE link handle for your IDE. Create an IDE link handle before you use the remove method. .</p> <p>filename</p> <p>Replace <i>filename</i> with the name of the file you are removing, or the source file from which you are removing debug points. If the file is not located in the active project, MATLAB returns a warning instead of completing the action.</p> <p>filetype</p> <p>To remove a project, enter 'project'. To remove a source file, enter 'text'.</p> <p>Default: 'text'</p>

addr

Enter the memory address of the debug point. Enter 'all' to remove the breakpoints.

debugtype

Enter the type of debug point to remove. The IDE provide several types of debug points. Refer to the IDE help documentation for information on their respective behavior.

Default: 'break' (breakpoint)

line

Enter the line number of the debug point located in a file.

timeout

Enter a time limit, in seconds, for the method to complete an action.

Examples

After you have a project in the IDE, you can delete files from it using `remove` from the MATLAB software command line. For example, build a project and load the resulting `.out` file. With the project build complete, load your `.out` file by typing

```
IDE_Obj.load('filename.out')
```

Now remove one file from your project

```
IDE_Obj.remove('filename')
```

You see in the IDE that the file no longer appears.

See Also

`add` | `open`

restart

Purpose Reload most recent program file to processor signal processor

Syntax `IDE_Obj.restart`
`IDE_Obj.restart(timeout)`

IDEs This function supports the following IDEs:

Description `IDE_Obj.restart` issues a restart command in the IDE debugger. The behavior of the restart process depends on the processor. Refer to the documentation for your IDE for details about using restart with various processors.

When `IDE_Obj` is an array that contains more than one processor, each processor calls `restart` in sequence.

`IDE_Obj.restart(timeout)` adds the optional `timeout` input argument. `timeout` defines an upper limit in seconds on the period the restart routine waits for completion of the restart process. If the time-out period is exceeded, `restart` returns control to MATLAB with a time-out error. In general, `restart` causes the processor to initiate a restart, even if the time-out period expires. The time-out error indicates that the restart confirmation was not received before the time-out period elapsed.

See Also `halt` | `isrunning` | `run`

Purpose Global model parameter structure

Syntax
`rsimgetrtp('model')`
`rsimgetrtp('model', 'AddTunableParamInfo', 'value')`

Description `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.

`rsimgetrtp('model', 'AddTunableParamInfo', 'value')` includes tunable parameter information in the parameter structure if *value* is 'on'. The function omits tunable parameters if *value* is 'off'. To use `AddTunableParamInfo`, you must enable inline parameters.

The model parameter structure contains the following fields:

Field	Description
<code>modelChecksum</code>	A four-element vector that encodes the structure. The Simulink Coder software 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> will 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, you must regenerate code for the model.
<code>parameters</code>	A structure that defines model global parameters.

The `parameters` substructure includes the following fields:

Field	Description
dataTypeName	Name of the parameter data type, for example, double
dataTypeID	An internal data type identifier
complex	Value 1 if parameter values are complex and 0 if real
dtTransIdx	Internal use only
values	Vector of parameter values

If you set 'AddTunableParamInfo' to 'on', the function creates and then deletes *model.rtw* from your current working folder and includes a map substructure that has the following fields:

Field	Description
Identifier	Parameter name
ValueIndicies	Vector of indices to parameter values
Dimensions	Vector indicating parameter dimensions

Examples

Return global parameter structure for model *rtwdemo_rsimtf* to *param_struct*:

```
rtwdemo_rsimtf
param_struct = rsimgetrtp('rtwdemo_rsimtf')

param_struct =

    modelChecksum: [1.7165e+009 3.0726e+009 2.6061e+009
2.3064e+009]
    parameters: [1x1 struct]
```

See Also

`rsimsetrtpparam`

How To

- “Create a MAT-File That Includes a Model Parameter Structure”
- “Update a Block Diagram”
- “Inline parameters”
- “Block Creation”
- “Tune Parameters”

rsimsetrtpparam

Purpose 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

```
rtp = rsimsetrtpparam(rtp, 'paramName', paramValue)
```

Takes an rtP structure with tunable parameter information and sets the values associated with 'paramName' to be paramValue if possible. There can be more than one name-value pair.

```
rtP = rsimsetrtpparam( rtP, idx, 'paramName', paramValue )
```

The rsimsetrtpparam utility allows for defining the values of an existing rtP parameter structure.

Takes an rtP structure with tunable parameter information and sets the values associated with 'paramName' to be paramValue in the *idx*'th parameter set. There can be more than one name-value pair. If the rtP structure does not have *idx* parameter sets, the first set is copied and appended until there are *idx* parameter sets. Subsequently, the *idx*'th set is changed.

Input Arguments

rtP

A parameter structure that contains the sets of parameter names and their respective values.

idx

An index used to indicate the number of parameter sets in the rtP structure

paramValue

The value of the rtP parameter, paramName

paramName

The name of the parameter set to add to the rtp structure

Output Arguments**rtp**

An expanded rtp parameter structure that contains idx additional parameter sets defined by the rsimsetrtpparam function call.

Definitions

The rtp structure should match the format of the structure returned by rsimsetrtpparam(modelName).

Examples

- 1 Expand the number of parameter sets in the 'rtp' structure to 10.

```
rtp = rsimsetrtpparam(rtp, 10);
```

- 2 Add three parameter sets to the parameter structure, 'rtp'.

```
rtp = rsimsetrtpparam(rtp, idx, 'X1', iX1, 'X2', iX2, 'Num', iNum);
```

See Also

rsimgetrtpparam

rtw_precompile_libs

Purpose Build libraries within model without building model

Syntax `rtw_precompile_libs('model', build_spec)`

Description `rtw_precompile_libs('model', build_spec)` builds libraries within *model*, according to the `build_spec` arguments, and places the libraries in a precompiled folder.

Input Arguments

model

Character array. Name of the model containing the libraries that you want to build.

build_spec

Structure of field and value pairs that define a build specification; fields except `rtwmakecfgDirs` are optional:

Field	Value
<code>rtwmakecfgDirs</code>	Cell array of strings that names the folders containing <code>rtwmakecfg</code> files for libraries that you want to precompile. Uses the <code>Name</code> and <code>Location</code> elements of <code>makeInfo.library</code> , as returned by the <code>rtwmakecfg</code> function, to specify name and location of precompiled libraries. If you use the <code>TargetPreCompLibLocation</code> parameter to specify the library folder, it overrides the <code>makeInfo.library.Location</code> setting. The specified model must contain blocks that use precompiled libraries that the <code>rtwmakecfg</code> files specify. The template makefile (TMF)-to-makefile conversion generates the library rules only if the conversion needs the libraries.
<code>libSuffix</code> (optional)	String that specifies the suffix, including the file type extension, to append to the name of each library (for example, <code>.a</code> or <code>_vc.lib</code>). The string

Field	Value
	must include a period (.). Set the suffix with either this field or the <code>TargetLibSuffix</code> parameter. If you specify a suffix with both mechanisms, the <code>TargetLibSuffix</code> setting overrides the setting of this field.
<code>intOnlyBuild</code> (optional)	Boolean flag. When set to <code>true</code> , indicates the function optimizes the libraries so that they compile from integer code only. Applies to ERT-based targets only.
<code>makeOpts</code> (optional)	String that specifies an option to include in the <code>rtwMake</code> command line.
<code>addLibs</code> (optional)	Cell array of structures that specify the libraries to build that an <code>rtwmakecfg</code> function does not specify. Define each structure with two fields that are character arrays: <ul style="list-style-type: none"> • <code>libName</code> — name of the library without a suffix • <code>libLoc</code> — location for the precompiled library The TMF can specify other libraries and how to build them. Use this field if you must precompile libraries.

Examples

Build the libraries in `my_model` without building `my_model`:

```
% Specify the library suffix
if isunix
    suffix = '.a';
else
    suffix = '_vc.lib';
end
set_param(my_model, 'TargetLibSuffix', suffix);

% Set the prcompiled library folder
set_param(my_model, 'TargetPreCompLibLocation', fullfile(pwd, 'lib'));
```

rtw_precompile_libs

```
% Define a build specification that specifies the location of the files to compile.
build_spec = [];
build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};

% Build the libraries in 'my_model'
rtw_precompile_libs(my_model, build_spec);
```

How To

- “Precompile S-Function Libraries”
- “Recompile Precompiled Libraries”

Purpose

Initiate build process

Syntax

```
rtwbuild(model)
rtwbuild(model,Name,Value )

rtwbuild(subsystem)

rtwbuild(subsystem,'Mode','ExportFunctionCalls')
blockHandle =
rtwbuild(subsystem,'Mode','ExportFunctionCalls')
rtwbuild(subsystem,'Mode','ExportFunctionCalls',
    'ExportFunctionInitializeFunctionName', fcname)
```

Description

`rtwbuild(model)` generates code from `model` based on current model configuration parameter settings. If `model` is not already loaded into the MATLAB environment, `rtwbuild` loads it before generating code.

If you clear the **Generate code only** model configuration parameter, the function generates code and builds an executable image.

`rtwbuild(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`rtwbuild(subsystem)` generates code from `subsystem` based on current model configuration parameter settings. Before initiating the build, open (or load) the parent model.

If you clear the **Generate code only** model configuration parameter, the function generates code and builds an executable image.

`rtwbuild(subsystem,'Mode','ExportFunctionCalls')`, if you have an Embedded Coder software license, generates code from `subsystem` that includes function calls that you can export to external application code.

`blockHandle =`
`rtwbuild(subsystem, 'Mode', 'ExportFunctionCalls')`,
if you have an Embedded Coder license and **Code Generation > Verification > Create block** is set to SIL, returns the handle to a SIL block created for code generated from the specified subsystem. You can then use the SIL block for SIL verification testing.

`rtwbuild(subsystem, 'Mode', 'ExportFunctionCalls,`
`'ExportFunctionInitializeFunctionName', fcname)` names the exported initialization function, specified as a string, for the specified subsystem.

Input Arguments

model - Model for which to generate code or build an executable image

`handle` | `name`

Model for which to generate code or build an executable image, specified as a handle or string representing the model name.

Example: `'rtwdemo_export_functions'`

subsystem - Subsystem for which to generate code or build executable image

`name`

Subsystem for which to generate code or build an executable image, specified as a string representing the subsystem name or full block path.

Example: `'rtwdemo_export_functions/rtwdemo_subsystem'`

Data Types

`char`

Name-Value Pair Arguments

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

Example:

```
rtwbuild('rtwdemo_mdltreftop','ForceTopModelBuild',true)
```

'ForceTopModelBuild' - Force regeneration of top model code

false (default) | true

Force regeneration of top model code, specified as true or false.

If You Want to...	Specify...
Force the coder to regenerate code for the top model of a system that includes referenced models	true
Let the coder determine whether to regenerate top model code based on model and model parameter changes	false

Consider forcing regeneration of code for a top model if you make changes associated with external or custom code, such as code for a custom target. For example, you should set `ForceTopModelBuild` to true if you change

- TLC code
- S-function source code, including `rtwmakecfg.m` files
- Integrated custom code

Alternatively, you can force regeneration of top model code by deleting code generation folders, such as `s1prj` or the generated model code folder.

'OkayToPushNags' - Display build error messages in Diagnostic Viewer

false (default) | true

Display build error messages in Diagnostic Viewer, specified as true or false.

If You Want to...	Specify...
Display build error messages in the Diagnostic Viewer and in the Command Window	true
Display build error messages in the Command Window only	false

Output Arguments

blockHandle - Handle to SIL block created for generated subsystem code

handle

Handle to SIL block created for generated subsystem code. Returned only if both of the following conditions apply:

- You are licensed to use Embedded Coder software.
- The **Create block** parameter on the **Code Generation > Verification** pane of the Configuration Parameters dialog box is set to **SIL**.

Tips

You can initiate code generation and the build process by using the following options:

- Clear the **Generate code only** option on the **Code Generation** pane of the Configuration Parameters dialog box and click **Build**.
- Press **Ctrl+B**.
- Select **Code > C/C++ Code > Build Model**.
- Invoke the `slbuild` command from the MATLAB command line.

Examples

Generate Code and Build Executable Image for Model

Generate C code for model `rtwdemo_rtwintr`.

```
rtwbuild('rtwdemo_rtwintr')
```


For the GRT target, the coder generates the following code files and places them in folders rtwdemo_rtwintrg_grt_rtw and slprj/grt/_sharedutils.

Model Files	Shared Utility Files	Interface Files	Other Files
rtwdemo_rtwintrg.c rtwdemo_rtwintrg.h rtwdemo_rtwintrg_private.h rtwdemo_rtwintrgtypes.h	rtGetInf.c rtGetInf.h rtGetNaN.c rtGetNaN.h rt_nonfinite.c rt_nonfinite.h rtwtypes.h multiword_types.h builtin_typeid_types.h	rtmodel.h	rt_logging.c

If the following model configuration parameters settings apply, the coder generates additional results.

Parameter Setting	Results
Code Generation > Generate code only pane is cleared	Executable image rtwdemo_rtwintrg.exe
Code Generation > Report > Create code generation report is selected	Report appears, providing information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Force Top Model Build

Generate code and build an executable image for `rtwdemo_mdltreftop`, which refers to model `rtwdemo_mdltreftop`, regardless of model checksums and parameter settings.

```
rtwbuild('rtwdemo_mdltreftop','ForceTopModelBuild',true)
```

Display Error Messages in Diagnostic Viewer

Introduce an error to model `rtwdemo_mdltreftop` and save the model as `rtwdemo_mdltreftop_witherr`. Display build error messages in the Diagnostic Viewer and in the Command Window while generating code and building an executable image for model `rtwdemo_mdltreftop_witherr`.

```
rtwbuild('rtwdemo_mdltreftop_witherr','OkayToPushNags',true)
```

Generate Code and Build Executable Image for Subsystem

Generate C code for subsystem `Amplifier` in model `rtwdemo_rtwintr`.

```
rtwbuild('rtwdemo_rtwintr/Amplifier')
```

For the GRT target, the coder generates the following code files and places them in folders `Amplifier_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Utility Files	Interface Files	Other Files
Amplifier.c Amplifier.h Amplifier_private.h Amplifier_types.h	rtGetInf.c rtGetInf.h rtGetNaN.c rtGetNaN.h rt_nonfinite.c rt_nonfinite.h rtwtypes.h multiword_types.h builtin_typeid_types.h	rtmodel.h	rt_logging.c

If the following model configuration parameters settings apply, the coder generates additional results.

Parameter Setting	Results
Code Generation > Generate code only pane is cleared	Executable image Amplifier.exe
Code Generation > Report > Create code generation report is selected	Report appears, providing information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Build Subsystem for Exporting Code to External Application

Build an executable image from a function-call subsystem to export the image to external application code.

```
rtwdemo_export_functions
rtwbuild('rtwdemo_export_functions/rtwdemo_subsystem','Mode','ExportFunctionCalls')
```

The executable image `rtwdemo_subsystem.exe` appears in your working folder.

Create SIL Block for Verification

From a function-call subsystem, create a SIL block that you can use to test the code generated from a model.

Open subsystem `rtwdemo_subsystem` in model `rtwdemo_export_functions` and set **Code Generation > Verification > Create block** to **SIL**.

Create the SIL block.

```
mysilblockhandle = rtwbuild('rtwdemo_export_functions/rtwdemo_subsystem',...  
'Mode','ExportFunctionCalls')
```

The coder generates a SIL block for the generated subsystem code. You can add the block to an environment or test harness model that supplies test vectors or stimulus input. You can then run simulations that perform SIL tests and verify that the generated code in the SIL block produces the same result as the original subsystem.

Name Exported Initialization Function

Name the initialization function generated when building an executable image from a function-call subsystem.

```
rtwdemo_export_functions  
rtwbuild('rtwdemo_export_functions/rtwdemo_subsystem',...  
'Mode','ExportFunctionCalls','ExportFunctionInitializeFunctionName','subsysinit')
```

The initialization function name `subsysinit` appears in `rtwdemo_subsystem_ert_rtw/ert_main.c`.

See Also

`slbuild`

Concepts

- Initiate the Build Process
- “Program Builds”
- Control Regeneration of Top Model Code
- “Export Function-Call Subsystems”
- “Software-in-the-Loop (SIL) Simulation”

Purpose

Build folder information for model

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 significantly 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.

Note This function can return build folder information for protected models.

Input Arguments**model - Input data**

character string

Character string specifying the name of a Simulink model.

Example: 'sldemo_fuelsys'

Data Types

char

Output Arguments**folderstruct - Output data**

structure

Structure containing the following:

RTW.getBuildDir

Field	Description
BuildDirectory	String specifying fully qualified path to build folder for model.
CacheFolder	String specifying root folder in which to place model build artifacts used for simulation.
CodeGenFolder	String specifying root folder in which to place Simulink Coder™ code generation files.
RelativeBuildDir	String specifying build folder relative to the current working folder (pwd).
BuildDirSuffix	String specifying suffix appended to model name to create build folder.
ModelRefRelativeRootSimDir	String specifying the relative root folder for the model reference target simulation folder.
ModelRefRelativeRootTgtDir	String specifying the relative root folder for the model reference target build folder.
ModelRefRelativeBuildDir	String specifying model reference target build folder relative to current working folder (pwd).
ModelRefRelativeSimDir	String specifying model reference target simulation folder relative to current working folder (pwd).
ModelRefRelativeHdlDir	String specifying model reference target HDL folder relative to current working folder (pwd).
ModelRefDirSuffix	String specifying suffix appended to system target file name to create model reference build folder.
SharedUtilsSimDir	String specifying the shared utility folder for simulation.
SharedUtilsTgtDir	String specifying the shared utility folder for code generation.

Examples

Display Build Folder Information

```
>> 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 build folder information for the model MyModel.

```
>> folderstruct = RTW.getBuildDir('MyModel')
```

```
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'
```

RTW.getBuildDir

```
ModelRefDirSuffix: ''  
SharedUtilsSimDir: 'slprj\sim\_sharedutils'  
SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

See Also

rtwbuild

Concepts

- “Working Folder”
- “Folders Used During the Build Process”
- “Control the Location for Generated Files”

Purpose	Rebuild generated code	
Syntax	<pre>rtwrebuild() rtwrebuild('model') rtwrebuild('path')</pre>	
Description	<p><code>rtwrebuild()</code> recompiles generated code files you modified by invoking the makefile generated during the previous build. If you omit the input arguments, the current working folder must be the build folder of the model (not the model location).</p> <p>Use <code>rtwrebuild('model')</code> if your current working folder is one level above the build folder of the model (<code>pwd</code> when you initiated the Simulink Coder build).</p> <p>Use <code>rtwrebuild('path')</code> to specify the path to the build folder of the model.</p> <p>If your model includes referenced models, the Simulink Coder software builds the referenced models recursively before rebuilding the top model.</p>	
Input Arguments	<i>model</i>	String specifying the model name.
	<i>path</i>	String specifying the fully qualified path to the build folder for the model.
Examples	<p>Rebuild generated code for a model located in the current working folder (one level above its build folder):</p> <pre>rtwrebuild('mymodel')</pre> <p>Rebuild generated code for a model by specifying a path to its build folder:</p> <pre>rtwrebuild(fullfile('C:', 'work', 'mymodel_grt_rtw'))</pre>	

How To

- “Rebuild a Model”

Purpose Create generated code report for model with Simulink Report Generator

Syntax `rtwreport(model)`
`rtwreport(model, folder)`

Description `rtwreport(model)` creates a report of code generation information for a model. Before creating the report, the function loads the model and generates code. The Simulink Coder software 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 Simulink code generation folder, `slprj`, must reside in the parent folder of `folder`. If the software cannot find the folder, an error occurs and code is not generated.

Input Arguments

model - Model name

string

Model name for which the report is generated, specified as a string.

Example: `'rtwdemo_secondOrderSystem'`

Data Types

char

folder - Build folder name

string

Build folder name, specified as a string. When you have multiple build folders, include a folder name. For example, if you have multiple builds using different targets, such as GRT and ERT.

Example: 'rtwdemo_secondOrderSystem_grt_rtw'

Data Types

char

Examples

Create Report Documenting Generated Code

Create a report for model `rtwdemo_secondOrderSystem`:

```
rtwreport('rtwdemo_secondOrderSystem');
```

Create Report Specifying Build Folder

Create a report for model `rtwdemo_secondOrderSystem` using build folder, `rtwdemo_secondOrderSystem_grt_rtw`:

```
rtwreport('rtwdemo_secondOrderSystem', 'rtwdemo_secondOrderSystem_grt_rtw');
```

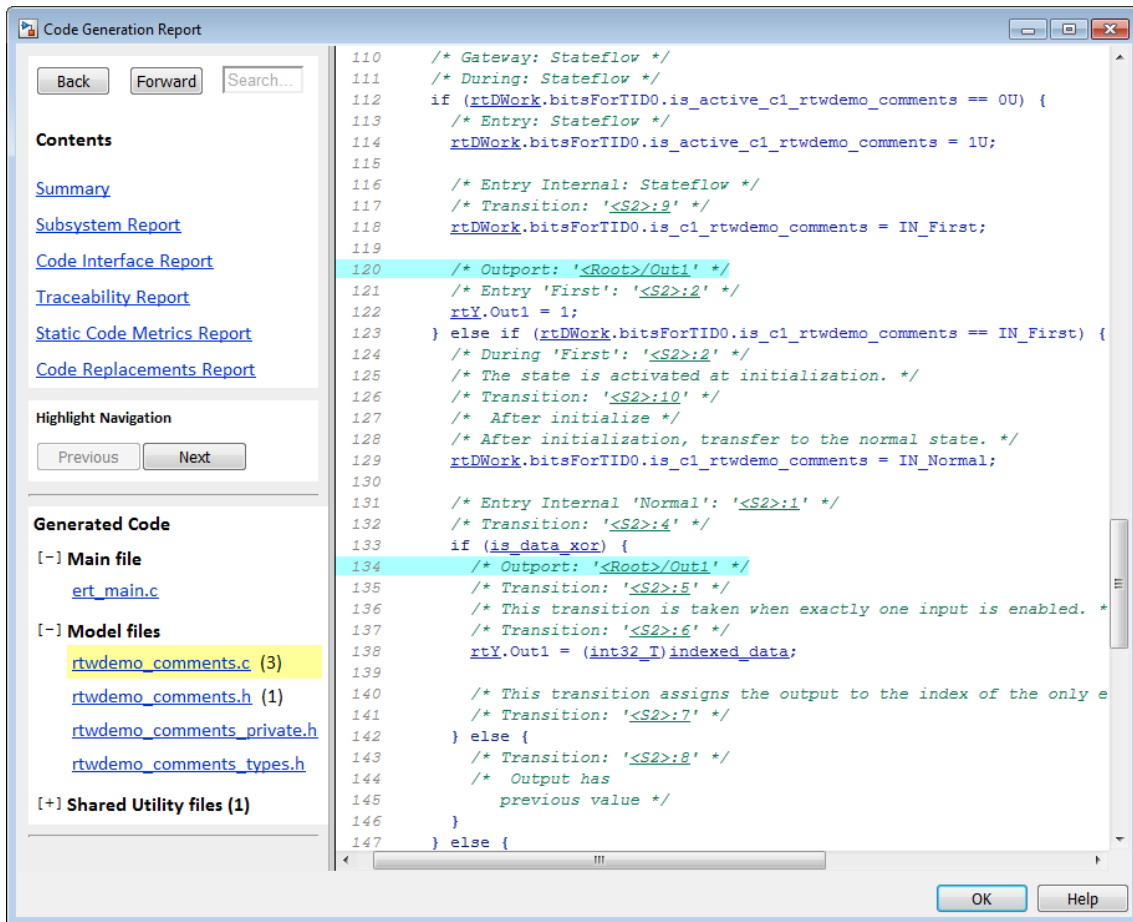
Related Examples

- “Document Generated Code with Simulink Report Generator™”
- Import Generated Code

Concepts

- “Report Explorer”
- Code Generation Summary

Purpose	Trace a block to generated code in HTML code generation report
Syntax	<code>rtwtrace('blockpath')</code>
Description	<p><code>rtwtrace('blockpath')</code> opens an HTML code generation report that displays contents of the source code file, and highlights the line of code corresponding to the specified block.</p> <p>Before calling <code>rtwtrace</code>, make sure:</p> <ul style="list-style-type: none">• You select an ERT-based model and enabled model to code navigation. To do this, on the Configuration Parameters dialog box, select the Code Generation > Report pane, and select the Model-to-code parameter.• You generate code for the model using the Embedded Coder software.• You have the build folder under the current working folder; otherwise, <code>rtwtrace</code> may produce an error.
Input Arguments	<p>blockpath - block path string</p> <p><code>blockpath</code> is a string enclosed in quotes specifying the full Simulink block path, for example, '<i>model_name/block_name</i>'.</p> <p>Example: 'Out1'</p> <p>Data Types char</p>
Examples	<p>Display Generated Code for a Block</p> <p>Display the generated code for block Out1 in the model <code>rtwdemo_comments</code> in HTML code generation report:</p> <pre>rtwtrace('rtwdemo_comments/Out1')</pre>



Alternatives

To trace from a block in the model diagram, right-click a block and select **C/C++ Code > Navigate to C/C++ Code**.

Related Examples

- “Trace Model Objects to Generated Code”
- “Model-to-code” on page 4-59

Purpose Execute program loaded on processor

Syntax

```

IDE_Obj.run
IDE_Obj.run('runopt')
IDE_Obj.run(..., timeout)

```

IDEs This function supports the following IDEs:

Description *IDE_Obj*.run runs the program file loaded on the referenced processor, returning immediately after the processor starts running. Program execution starts from the location of program counter (PC). Usually, the program counter is positioned at the top of the executable file. However, if you stopped a running program with `halt`, the program counter may be anywhere in the program. `run` starts the program from the program counter current location.

If *IDE_Obj* references more than one processor, each processor calls `run` in sequence.

IDE_Obj.run('runopt') includes the parameter `runopt` that defines the action of the `run` method. The options for `runopt` are listed in the following table.

runopt string	Description
'run'	Executes the run and waits to confirm that the processor is running, and then returns to MATLAB.
'runtohalt'	Executes the run but then waits until the processor halts before returning. The halt can be the result of the PC reaching a breakpoint, or by direct interaction with the IDE, or by the normal program exit process.
'tohalt'	Waits until the running program has halted. Unlike the other options, this selection does not execute a run, it simply waits for the running program to halt.

runopt string	Description
'main'	This option resets the program and executes a run until the start of function 'main'.
'tofunc'	<p>This option must be followed by an extra parameter <i>funcname</i>, the name of the function to run to:</p> <pre>IDE_Obj.run('tofunc', funcname)</pre> <p>This executes a run from the present PC location until the start of function <i>funcname</i> is reached. If <i>funcname</i> is not along the program's normal execution path, <i>funcname</i> is not reached and the method times out.</p>

In the 'run' and 'runtohalt' cases, a halt can be caused by a breakpoint, a direct interaction with the IDE, or by a normal program exit.

The following table shows the availability of the *runopt* options by IDE.

	CCS IDE	MULTI IDE	VisualDSP++ IDE
'run'	Yes	Yes	Yes
'runtohalt'	Yes	Yes	Yes
'tohalt'	Yes	Yes	
'main'	Yes	Yes	
'tofunc'	Yes	Yes	

IDE_Obj.run(..., timeout) adds input argument *timeout*, to allow you to set the time out to a value different from the global timeout value. The *timeout* value specifies how long, in seconds, MATLAB waits for the processor to start executing the loaded program before returning.

Most often, the 'run' and 'runtohalt' options cause the processor to initiate execution, even when a timeout is reached. The timeout indicates that the confirmation was not received before the timeout period elapsed.

See Also

halt | load |

Simulink.fileGenControl

Purpose Specify root folders in which to put files generated by diagram updates and model builds

Syntax

```
Simulink.fileGenControl(action)
cfg = Simulink.fileGenControl('getConfig')
Simulink.fileGenControl('reset', 'keepPreviousPath', true)
Simulink.fileGenControl('setConfig', 'config', cfg,
    'keepPreviousPath', true, 'createDir', true)
Simulink.fileGenControl('set', 'CacheFolder',
    cacheFolderPath,
    'CodeGenFolder', codegenFolderPath, 'keepPreviousPath', true,
    'createDir', true)
```

Description `Simulink.fileGenControl(action)` performs a requested action related to the file generation control parameters `CacheFolder` and `CodeGenFolder` for the current MATLAB session. `CacheFolder` specifies the root folder in which to put model build artifacts used for simulation, and `CodeGenFolder` specifies the root folder in which to put Simulink Coder code generation files. The initial session defaults for these parameters are provided by the Simulink preferences “Simulation cache folder” and “Code generation folder”.

`cfg = Simulink.fileGenControl('getConfig')` returns a handle to an instance of the `Simulink.FileGenConfig` object containing the current values of the `CacheFolder` and `CodeGenFolder` parameters. You can then use the handle to get or set the `CacheFolder` and `CodeGenFolder` fields.

`Simulink.fileGenControl('reset', 'keepPreviousPath', true)` reinitializes the `CacheFolder` and `CodeGenFolder` parameters to the values provided by the Simulink preferences “Simulation cache folder” and “Code generation folder”. To keep the previous values of `CacheFolder` and `CodeGenFolder` in the MATLAB path, specify `'keepPreviousPath'` with the value `true`.

`Simulink.fileGenControl('setConfig', 'config', cfg, 'keepPreviousPath', true, 'createDir', true)` sets the file generation control configuration for the current MATLAB session

by passing a handle to an instance of the `Simulink.FileGenConfig` object containing values for the `CacheFolder` and/or `CodeGenFolder` parameters. To keep the previous values of `CacheFolder` and `CodeGenFolder` in the MATLAB path, specify `'keepPreviousPath'` with the value `true`. To create the specified file generation folders if they do not already exist, specify `'createDir'` with the value `true`.

`Simulink.fileGenControl('set', 'CacheFolder', cacheFolderPath, 'CodeGenFolder', codegenFolderPath, 'keepPreviousPath', true, 'createDir', true)` sets the file generation control configuration for the current MATLAB session by directly passing values for the `CacheFolder` and/or `CodeGenFolder` parameters. To keep the previous values of `CacheFolder` and `CodeGenFolder` in the MATLAB path, specify `'keepPreviousPath'` with the value `true`. To create the specified file generation folders if they do not already exist, specify `'createDir'` with the value `true`.

Input Arguments

action

String specifying one of the following actions:

Action	Description
<code>getConfig</code>	Returns a handle to an instance of the <code>Simulink.FileGenConfig</code> object containing the current values of the <code>CacheFolder</code> and <code>CodeGenFolder</code> parameters.
<code>reset</code>	Reinitializes the <code>CacheFolder</code> and <code>CodeGenFolder</code> parameters to the values provided by the Simulink preferences “Simulation cache folder” and “Code generation folder”.

Simulink.fileGenControl

Action	Description
set	Sets the CacheFolder and/or CodeGenFolder parameters for the current MATLAB session by directly passing values.
setConfig	Sets the CacheFolder and/or CodeGenFolder parameters for the current MATLAB session by passing a handle to an instance of the Simulink.FileGenConfig object.

'config', cfg

Specifies a handle *cfg* to an instance of the `Simulink.FileGenConfig` object containing values to be set for the `CacheFolder` and/or `CodeGenFolder` parameters.

'CacheFolder', cacheFolderPath

Specifies a string value *cacheFolderPath* representing a folder path to directly set for the `CacheFolder` parameter.

'CodeGenFolder', codeGenFolderPath

Specifies a string value *codeGenFolderPath* representing a folder path to directly set for the `CodeGenFolder` parameter.

Note You can specify absolute or relative paths to the 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'.
-

'keepPreviousPath', true

For reset, set, or setConfig, specifies whether to keep the previous values of CacheFolder and CodeGenFolder in the MATLAB path. If 'keepPreviousPath' is omitted or specified as false, the call removes previous folder values from the MATLAB path.

'createDir', true

For set or setConfig, specifies whether to create the specified file generation folders if they do not already exist. If 'createDir' is omitted or specified as false, the call throws an exception if a specified file generation folder does not exist.

Output Arguments

cfg

Handle to an instance of the Simulink.FileGenConfig object containing the current values of the CacheFolder and CodeGenFolder parameters.

Examples

Obtain the current CacheFolder and CodeGenFolder values:

```
cfg = Simulink.fileGenControl('getConfig');
```

Simulink.fileGenControl

```
myCacheFolder = cfg.CacheFolder;
myCodeGenFolder = cfg.CodeGenFolder;
```

Set the `CacheFolder` and `CodeGenFolder` parameters for the current MATLAB session by first setting fields in an instance of the `Simulink.FileGenConfig` object and then passing a handle to the object instance:

```
% Get the current configuration
cfg = Simulink.fileGenControl('getConfig');
% Change the parameters to C:\cachefolder and current working folder
cfg.CacheFolder = fullfile('C:', 'cachefolder');
cfg.CodeGenFolder = pwd;
Simulink.fileGenControl('setConfig', 'config', cfg);
```

Directly set the `CacheFolder` and `CodeGenFolder` parameters for the current MATLAB session without creating an instance of the `Simulink.FileGenConfig` object:

```
myCacheFolder = fullfile('C:', 'cachefolder');
myCodeGenFolder = pwd;
Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ...
    'CodeGenFolder', myCodeGenFolder);
```

Reinitialize the `CacheFolder` and `CodeGenFolder` parameters to the values provided by the Simulink preferences “Simulation cache folder” and “Code generation folder”:

```
Simulink.fileGenControl('reset');
```

Alternatives

Instead of setting the `CacheFolder` and `CodeGenFolder` parameters just for the current MATLAB session, you can set the Simulink preferences “Simulation cache folder” and “Code generation folder”, which provide the initial MATLAB session defaults. The preferences can be set using

the Simulink Preferences dialog box or using the MATLAB command `set_param`.

See Also

“Simulation cache folder” | “Code generation folder”

How To

- “Control the Location for Generated Files”

Simulink.ModelReference.protect

Purpose Obscure referenced model contents to hide intellectual property

Syntax

```
Simulink.ModelReference.protect(model)
Simulink.ModelReference.protect(model,Name,Value)

[harnessHandle] =
Simulink.ModelReference.protect(model, 'Harness',
    true)
[~,neededVars] = Simulink.ModelReference.protect(model)
```

Description

`Simulink.ModelReference.protect(model)` creates a protected model from the specified `model`. It places the protected model in the current working folder. The protected model has the same name as the source model. It has the extension `.slxp`.

`Simulink.ModelReference.protect(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[harnessHandle] = Simulink.ModelReference.protect(model, 'Harness', true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~,neededVars] = Simulink.ModelReference.protect(model)` returns a cell array that includes the names of base workspace variables used by the protected model.

Input Arguments

model - Model name
string (default)

Model name, specified as a string. It contains the name of a model or the path name of a Model block that references the model to be protected.

Name-Value Pair Arguments

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

Example:

```
'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true
```

specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

'Harness' - Option to create a harness model

false (default) | true

Option to create a harness model, specified as a Boolean value.

Example: `'Harness',true`

'Mode' - Model protection mode

'Normal' (default) | 'Accelerator' | 'CodeGeneration' | 'ViewOnly'

Model protection mode, specified as a string. Specify one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'` or `'Accelerator'` mode.
- `'CodeGeneration'`: The top model can run in `'Normal'` or `'Accelerator'` mode and support code generation.
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. Turns on the read-only view mode.

Example: `'Mode','Accelerator'`

'ObfuscateCode' - Option to obfuscate generated code

true (default) | false

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation during protection is enabled.

Example: 'ObfuscateCode',true

'Path' - Folder for protected model

current working folder (default) | string

Folder for protected model, specified as a Boolean value.

Example: 'Path', 'C:\Work'

'Report' - Option to generate a report

false (default) | true

Option to generate a report, specified as a Boolean value.

To view the report, double-click the protected model block. The report is generated in HTML format. It includes environment information and the model interface.

Example: 'Report',true

'OutputFormat' - Protected code visibility

'CompiledBinaries' (default) | 'MinimalCode' |
'AllReferencedHeaders'

Note This argument affects the output only when you specify Mode as 'Accelerator' or 'CodeGeneration'. When you specify Mode as 'Normal', only a MEX-file is part of the output package.

Protected code visibility, specified as a string. This argument determines what part of the code generated for a protected model is visible to users. Specify one of the following values:

- 'CompiledBinaries': Only binary files and headers are visible.

- 'MinimalCode': All code in the build directory is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- 'AllReferencedHeaders': All code in the build folder is visible. All headers referenced by the code are also visible.

Example: 'OutputFormat', 'AllReferencedHeaders'

'Webview' - Option to include a Web view

false (default) | true

Option to include a read-only view of protected model, specified as a Boolean value.

Example: 'Webview', true

'Encrypt' - Option to encrypt protected model

false (default) | true

Option to encrypt a protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model:
`Simulink.ModelReference.ProtectedModel.setPasswordForView`
- Password for simulation:
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`
- Password for code generation:
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`

Example: 'Encrypt', true

'CustomPostProcessingHook' - Option to add postprocessing function for protected model files

function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an

Simulink.ModelReference.protect

input variable. This object provides information on the source code files and other files generated during protected model creation. It also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

Example:

```
'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)
```

Output Arguments

harnessHandle - Handle of the harness model

double

Handle of the harness model, returned as a double or 0, depending on the value of Harness.

If Harness is true, the value is the handle of the harness model; otherwise, the value is 0.

neededVars - Names of base workspace variables

cell array

Names of base workspace variables used by the model being protected, returned as a cell array.

The cell array can also include variables that the protected model does not use.

Examples

Protect Referenced Model

Protect a referenced model and place the protected model in the current working folder.

```
sldemo_mdhref_bus;  
model= 'sldemo_mdhref_counter_bus'
```

```
Simulink.ModelReference.protect(model);
```

A protected model named `sldemo_mdhref_counter_bus.slxp` is created. The file is placed in the current working folder.

Place Protected Model in Specified Folder

Protect a referenced model and place the protected model in a specified folder.

```
sldemo_mdhref_bus;  
model= 'sldemo_mdhref_counter_bus'
```

```
Simulink.ModelReference.protect(model, 'Path', 'C:\Work');
```

A protected model named `sldemo_mdhref_counter_bus.slxp` is created. The file is placed in `C:\Work`.

Generate Code for Protected Model

Protect a referenced model, generate code for it in Normal mode, and obfuscate the code.

```
sldemo_mdhref_bus;  
model= 'sldemo_mdhref_counter_bus'
```

```
Simulink.ModelReference.protect(model, 'Path', 'C:\Work', 'Mode', 'CodeGeneration', 'Obfuscate
```

A protected model named `sldemo_mdhref_counter_bus.slxp` is created. The file is placed in the `C:\Work` folder. The protected model runs as a child of the parent model. The code generated for the protected model is obfuscated.

Control Code Visibility for Protected Model

Control code visibility by allowing users to view only binary files and headers in the code generated for a protected model.

```
sldemo_mdhref_bus;  
model= 'sldemo_mdhref_counter_bus'
```

```
Simulink.ModelReference.protect(model, 'Mode', 'CodeGeneration', 'OutputFormat', 'CompiledBin
```

A protected model named `sldemo_mdhref_counter_bus.slxp` is created. The file is placed in the current working folder. Users can view

Simulink.ModelReference.protect

only binary files and headers in the code generated for the protected model.

Create Harness Model for Protected Model

Create a harness model for a protected model and generate an HTML report.

```
sldemo_md1ref_bus;  
modelPath= 'sldemo_md1ref_bus/CounterA'  
  
[harnessHandle] = Simulink.ModelReference.protect(modelPath, 'Path', 'C:\Work', 'Harness', true, '
```

A protected model named `sldemo_md1ref_counter_bus.slxp` is created, along with an untitled harness model. The protected model file is placed in the `C:\Work` folder. The folder also contains an HTML report. The handle of the harness model is returned in `harnessHandle`.

Alternatives

“Create a Protected Model”

Related Examples

- Protected Models for Model Reference
- “Test the Protected Model”
- “Package a Protected Model”
- “Specify Custom Obfuscator for Protected Model”

Concepts

- “Protected Model”
- “Protect a Referenced Model”
- “Protected Model File”
- “Harness Model”
- “Protected Model Report”
- “Code Generation Support in a Protected Model”

Simulink.ModelReference.ProtectedModel.HookInfo

Purpose Represent files and exported symbols generated by creation of protected model

Description Specifies information about files and symbols generated when creating a protected model. The creator of a protected model can use this information for postprocessing of the generated files prior to packaging. Information includes:

- List of source code files (*.c, *.h, *.cpp, *.hpp).
- List of other related files (*.mat, *.rsp, *.prj, etc.).
- List of exported symbols that you must not modify.

Construction To access the properties of this class, use the 'CustomPostProcessingHook' option of the Simulink.ModelReference.protect function. The value for the option is a handle to a postprocessing function accepting a Simulink.ModelReference.ProtectedModel.HookInfo object as input.

Properties **ExportedSymbols - Exported Symbols**

cell array of strings

A list of exported symbols generated by protected model that you must not modify. Default value is empty.

NonSourceFiles - Non source code files

cell array of strings

A list of non-source files generated by protected model creation. Examples are *.mat, *.rsp, and *.prj. Default value is empty.

SourceFiles - Source code files

cell array of strings

A list of source code files generated by protected model creation. Examples are *.c, *.h, *.cpp, and *.hpp. Default value is empty.

Simulink.ModelReference.ProtectedModel.HookInfo

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

See Also `Simulink.ModelReference.protect`

Related Examples

- “Specify Custom Obfuscator for Protected Model”

Purpose	Return current value for custom target configuration option
Syntax	<code>value = slConfigUIGetVal(hDlg, hSrc, 'OptionName')</code>
Input Arguments	<p><code>hDlg</code> Handle created in the context of a <code>SelectCallback</code> 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.</p> <p><code>hSrc</code> Handle created in the context of a <code>SelectCallback</code> 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.</p> <p><code>'OptionName'</code> Quoted name of the TLC variable defined for a custom target configuration option.</p>
Output Arguments	Current value of the specified option. The data type of the return value depends on the data type of the option.
Description	The <code>slConfigUIGetVal</code> function is used in the context of a user-written <code>SelectCallback</code> function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use <code>slConfigUIGetVal</code> to read the current value of a specified target option.
Examples	<p>In the following example, the <code>slConfigUIGetVal</code> function returns the value of the Terminate function required option on the Code Generation > Interface pane of the Configuration Parameters dialog box.</p> <pre>function usertarget_selectcallback(hDlg, hSrc) disp(['*** Select callback triggered:', sprintf('\n'), ... ' Uncheck and disable "Terminate function required."]);</pre>

slConfigUIGetVal

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

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

See Also

slConfigUISetEnabled | slConfigUISetVal

How To

- “Define and Display Custom Target Options”
- “Parameter Command-Line Information Summary” on page 4-375
- “Support Optional Features”

Purpose	Enable or disable custom target configuration option
Syntax	<pre>slConfigUISetEnabled(hDlg, hSrc, 'OptionName', true) slConfigUISetEnabled(hDlg, hSrc, 'OptionName', false)</pre>
Arguments	<p>hDlg Handle created in the context of a <code>SelectCallback</code> 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.</p> <p>hSrc Handle created in the context of a <code>SelectCallback</code> 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.</p> <p>'OptionName' Quoted name of the TLC variable defined for a custom target configuration option.</p> <p>true Specifies that the option should be enabled.</p> <p>false Specifies that the option should be disabled.</p>
Description	<p>The <code>slConfigUISetEnabled</code> function is used in the context of a user-written <code>SelectCallback</code> function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use <code>slConfigUISetEnabled</code> to enable or disable a specified target option.</p> <p>If you use this function to disable a parameter that is represented in the Configuration Parameters dialog box, the parameter appears greyed out in the dialog context.</p>
Examples	<p>In the following example, the <code>slConfigUISetEnabled</code> function disables the Terminate function required option on the Code Generation > Interface pane of the Configuration Parameters dialog box.</p>

slConfigUISetEnabled

```
function usertarget_selectcallback(hDlg, hSrc)

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

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

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

See Also

slConfigUIGetVal | slConfigUISetVal

How To

- “Define and Display Custom Target Options”
- “Parameter Command-Line Information Summary” on page 4-375
- “Support Optional Features”

Purpose	Set value for custom target configuration option
Syntax	<code>slConfigUISetVal(hDlg, hSrc, 'OptionName', OptionValue)</code>
Arguments	<p>hDlg Handle created in the context of a <code>SelectCallback</code> 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.</p> <p>hSrc Handle created in the context of a <code>SelectCallback</code> 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.</p> <p>'OptionName' Quoted name of the TLC variable defined for a custom target configuration option.</p> <p>OptionValue Value to be set for the specified option.</p>
Description	The <code>slConfigUISetVal</code> function is used in the context of a user-written <code>SelectCallback</code> function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use <code>slConfigUISetVal</code> to set the value of a specified target option.
Examples	In the following example, the <code>slConfigUISetVal</code> function sets the value 'off' for the Terminate function required option on the Code Generation > Interface pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

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

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

slConfigUISetVal

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

See Also

slConfigUIGetVal | slConfigUISetEnabled

How To

- “Define and Display Custom Target Options”
- “Parameter Command-Line Information Summary” on page 4-375
- “Support Optional Features”

Purpose

Select target for configuration set

Syntax

```
switchTarget(myConfigObj,systemTargetFile,[])  
switchTarget(myConfigObj,systemTargetFile,targetOptions)
```

Description

`switchTarget(myConfigObj,systemTargetFile,[])` selects a system target file for the active configuration set.

`switchTarget(myConfigObj,systemTargetFile,targetOptions)` sets the configuration parameters specified by `targetOptions`.

Input Arguments

myConfigObj - Input data

configuration set object

A configuration set object of `Simulink.ConfigSet` or configuration reference object of `Simulink.ConfigSetRef`. Call `getActiveConfigSet` to get the configuration set object.

Example: `myConfigObj = getActiveConfigSet(model);`

systemTargetFile - Input data

name of system target file

Specify the name of the system target file, such as `ert.tlc` for Embedded Coder, or `grt.tlc` for Generic Real-Time Target coder.

Example: `systemTargetFile = `ert.tlc`;`

Data Types

char

targetOptions - Input options

structure of configuration parameter options

You can choose to modify certain configuration parameters by filling in values in a structure for fields listed below. If you do not want to use options, specify an empty structure(`[]`).

switchTarget

Field	Value
TemplateMakefile	String specifying file name of template makefile.
TLCOptions	String specifying TLC argument.
MakeCommand	String specifying make command MATLAB language file.
Description	String specifying description of target.

Example: targetOptions.TemplateMakefile = 'grt_default_tmf';

Data Types
struct

Examples

Select target file without options

```
% Get the active configuration set for 'model'  
myConfigObj = getActiveConfigSet(model);  
% Change the system target file for the configuration set.  
switchTarget(myConfigObj,'ert.tlc',[]);
```

Select target file with options

```
>> % Get the active configuration set for the 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';  
>>  
>> % Verify values (optional)  
>> targetOptions
```

```
TLCOptions: '-aVarName=1'
```



```
        MakeCommand: 'make_rtw'  
        Description: 'my target'  
        TemplateMakefile: 'grt_default_tmf'  
  
>> % Define a system target file  
>> targetSystemFile='grt.tlc';  
>>  
>> % Change the system target file and target options  
>> % for the configuration set  
>> switchTarget(myConfigObj,targetSystemFile,targetOptions);
```

See Also

[getActiveConfigSet](#) | [Simulink.ConfigSet](#) |
[Simulink.ConfigSetRef](#)

Concepts

- “Select a System Target File Programmatically”
- “Select a Target”
- “Set Target Language Compiler Options”

Purpose Invoke Target Language Compiler to convert model description file to generated code

Syntax `tlc [-options] [file]`

Description `tlc` invokes the Target Language Compiler (TLC) from the command prompt. The TLC converts the model description file, `model.rtw` (or similar files), into target-specific code or text. Typically, you do not call this command because the Simulink Coder build process automatically invokes the Target Language Compiler when generating code. For more information, see “Introduction to the Target Language Compiler”.

Note This command is used only when invoking the TLC separately from the Simulink Coder build process. You cannot use this command to initiate code generation for a model.

`tlc [-options] [file]`

You can change the default behavior by specifying one or more compilation *options* as described in “Options” on page 2-172

Options

You can specify one or more compilation options with each `tlc` command. Use spaces to separate options and arguments. TLC resolves options from left to right. If you use conflicting options, the rightmost option prevails. The `tlc` options are:

- “-r Specify Simulink® Coder™ filename” on page 2-173
- “-v Specify verbose level” on page 2-173
- “-l Specify path to local include files” on page 2-173
- “-m Specify maximum number of errors” on page 2-173
- “-O Specify the output file path” on page 2-173
- “-d[a|c|n|o] Invoke debug mode” on page 2-173

- “-a Specify parameters” on page 2-174
- “-p Print progress” on page 2-174
- “-lint Performance checks and runtime statistics” on page 2-174
- “-xO Parse only” on page 2-174

-r Specify Simulink Coder filename

-r file_name

Specify the filename that you want to translate.

-v Specify verbose level

-v number

Specify a number indicating the verbose level. If you omit this option, the default value is one.

-l Specify path to local include files

-l path

Specify a folder path to local include files. The TLC searches this path in the order specified.

-m Specify maximum number of errors

-m number

Specify the maximum number of errors reported by the TLC prior to terminating the translation of the `.t1c` file.

If you omit this option, the default value is five.

-O Specify the output file path

-O path

Specify the folder path to place output files.

If you omit this option, TLC places output files in the current folder.

-d[a|c|n|o] Invoke debug mode

-da execute any `%assert` directives

-dc invoke the TLC command line debugger

-dn produce log files, which indicate those lines hit and those lines missed during compilation.

-do disable debugging behavior

-a Specify parameters

-a *identifier = expression*

Specify parameters to change the behavior of your TLC program. For example, this option is used by the Simulink Coder software to set inlining of parameters or file size limits.

-p Print progress

-p *number*

Print a '.' indicating progress for every number of TLC primitive operations executed.

-lint Performance checks and runtime statistics

-lint

Perform simple performance checks and collect runtime statistics.

-xO Parse only

-xO

Parse only a TLC file; do not execute it.

Purpose	Update files in model build information with missing paths and file extensions
Syntax	<code>updateFilePathsAndExtensions(<i>buildinfo</i>, <i>extensions</i>)</code> <i>extensions</i> is optional.
Arguments	<i>buildinfo</i> Build information returned by <code>RTW.BuildInfo</code> . <i>extensions</i> (optional) A cell array of character arrays that specifies the extensions (file types) of files for which to search and include in the update processing. By default, the function searches for files with a <code>.c</code> extension. 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 <code>{'.c' '.cpp'}</code> and a folder contains <code>myfile.c</code> and <code>myfile.cpp</code> , an instance of <code>myfile</code> would be updated to <code>myfile.c</code> .
Description	Using paths that already exist in the model build information, the <code>updateFilePathsAndExtensions</code> function checks whether file references in the build information need to be updated with a path or file extension. This function can be particularly useful for <ul style="list-style-type: none">• Maintaining build information for a toolchain that requires the use of file extensions• Updating multiple customized instances of build information for a given model

Note If you need to use `updateFilePathsAndExtensions`, you should call it once, after you add files to the build information, to minimize the potential performance impact of the required disk I/O.

updateFilePathsAndExtensions

Examples

Create the folder path etcproj/etc in your working folder, add files etc.c, test1.c, and test2.c to the folder etc. This example assumes the working folder is w:\work\BuildInfo. From the working folder, update build information myModelBuildInfo with missing paths or file extensions.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, fullfile(pwd,...
    'etcproj', '/etc'), 'test');
addSourceFiles(myModelBuildInfo, {'etc' 'test1'...
    'test2'}, '', 'test');
before=getSourceFiles(myModelBuildInfo, true, true);
before
```

```
before =
```

```
    '\etc'    '\test1'    '\test2'
```

```
updateFilePathsAndExtensions(myModelBuildInfo);
after=getSourceFiles(myModelBuildInfo, true, true);
after{:}
```

```
ans =
```

```
w:\work\BuildInfo\etcproj\etc\etc.c
```

```
ans =
```

```
w:\work\BuildInfo\etcproj\etc\test1.c
```

```
ans =
```

```
w:\work\BuildInfo\etcproj\etc\test2.c
```

See Also

[addIncludeFiles](#) | [addIncludePaths](#) | [addSourceFiles](#) | [addSourcePaths](#) | [updateFileSeparator](#)

How To

- “Customize Post-Code-Generation Build Processing”

updateFileSeparator

Purpose	Change file separator used in model build information
Syntax	<code>updateFileSeparator(<i>buildinfo</i>, <i>separator</i>)</code>
Arguments	<i>buildinfo</i> Build information returned by <code>RTW.BuildInfo</code> . <i>separator</i> A character array that specifies the file separator \ (Windows®) or / (UNIX®) to be applied to file path specifications.
Description	<p>The <code>updateFileSeparator</code> function changes instances of the current file separator (/ or \) in the model build information to the specified file separator.</p> <p>The default value for the file separator matches the value returned by the MATLAB command <code>filesep</code>. For makefile based builds, you can override the default by defining a separator with the <code>MAKEFILE_FILESEP</code> macro in the template makefile (see “Cross-Compile Code Generated on Microsoft® Windows”). If the <code>GenerateMakefile</code> parameter is set, the Simulink Coder software overrides the default separator and updates the model build information after evaluating the <code>PostCodeGenCommand</code> configuration parameter.</p>
Examples	<p>Update object <code>myModelBuildInfo</code> to apply the Windows file separator.</p> <pre>myModelBuildInfo = RTW.BuildInfo; updateFileSeparator(myModelBuildInfo, '\\');</pre>
See Also	<code>addIncludeFiles</code> <code>addIncludePaths</code> <code>addSourceFiles</code> <code>addSourcePaths</code> <code>updateFilePathsAndExtensions</code>
How To	<ul style="list-style-type: none">• “Customize Post-Code-Generation Build Processing”• “Cross-Compile Code Generated on Microsoft Windows”

Purpose Write data to processor memory block

Syntax

```
mem=IDE_Obj.write(address,data)
mem=write(...,datatype)
mem=IDE_Obj.write(...,memorytype)
mem=IDE_Obj.write(...,timeout)
```

IDEs This function supports the following IDEs:

Description `mem=IDE_Obj.write(address,data)` writes *data*, a collection of values, to the memory space of the DSP processor referenced by `IDE_Obj`.

The *data* argument is a scalar, vector, or array of values to write to the memory of the processor. The block to write begins from the DSP memory location given by the input parameter *address*.

The method writes the data starting from *address* without regard to type-alignment boundaries in the DSP. Conversely, the byte ordering of the data type is automatically applied.

Note You cannot write data to processor memory while the processor is running.

The *address* argument is a decimal or hexadecimal representation of a memory address in the processor. The full memory address consist of two parts: the start address and the memory type. The memory type value can be explicitly defined using a numeric vector representation of the address.

Alternatively, the `IDE_Obj` object has a default memory type value which is applied if the memory type value is not explicitly incorporated into the passed address parameter. In DSP processors with only a single memory type, by setting the `IDE_Obj` object memory type value to zero it is possible to specify the addresses using the abbreviated (implied memory type) format.

You provide the *address* argument either as a numerical value that is a decimal representation of the DSP memory address, or as a string that `write` converts to the decimal representation of the start address. (Refer to function `hex2dec` in the *MATLAB Function Reference* that `read` uses to convert the hexadecimal string to a decimal value).

The following examples show how `write` uses the *address* argument.

address Parameter Value	Description
131082	Decimal address specification. The memory start address is 131082 and memory type is 0. This action is the same as specifying [131082 0].
[131082 1]	Decimal address specification. The memory start address is 131082 and memory type is 1.
'2000A'	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify *address* as cell array, in which case you can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1} = 131072; myaddress1{2} =  
'Program(PM) Memory';  
myaddress2 myaddress2{1} = '20000'; myaddress2{2} =  
'Program(PM) Memory';  
myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;
```

`mem=write(...,datatype)` where the *datatype* argument defines the interpretation of the raw values written to DSP memory. The *datatype* argument specifies the data format of the raw memory image. The data is written starting from *address* without regard to data type alignment boundaries in the DSP. The byte ordering of the data type

is automatically applied. The following MATLAB data types are supported.

MATLAB Data Type	Description
double	IEEE double-precision floating point value
single	IEEE single-precision floating point value
uint8	8-bit unsigned binary integer value
uint16	16-bit unsigned binary integer value
uint32	32-bit unsigned binary integer value
int8	8-bit signed two's complement integer value
int16	16-bit signed two's complement integer value
int32	32-bit signed two's complement integer value

`write` does not coerce data type alignment. Some combinations of *address* and *datatype* will be difficult for the processor to use.

`mem=IDE_Obj.write(...,memorytype)` adds an optional *memorytype* argument. Object `IDE_Obj` has a default memory type value 0 that `write` applies if the memory type value is not explicitly incorporated into the passed address parameter. In processors with only a single memory type, it is possible to specify the addresses using the implied memory type format by setting the value of the `IDE_Obj` *memorytype* property to zero.

`mem=IDE_Obj.write(...,timeout)` adds the optional *timeout* argument, which the number of seconds MATLAB waits for the write process to

write

complete. If the *timeout* period expires before the write process returns a completion message, MATLAB throws an error and returns. Usually the process works in spite of the error message.

Using write with VisualDSP++ IDE

Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
'program(pm) memory'	0	Blackfin and SHARC
'data(dm) memory'	1	SHARC
'data(dm) short word memory'	2	SHARC
'external data(dm) byte memory'	3	SHARC
'boot(prom) memory'	4	SHARC

Examples

Example with VisualDSP++ IDE

These three syntax examples show how to use `write` in some common ways. In the first example, write an array of 16-bit integers to location [131072 1].

```
IDE_Obj.write([131072 1],int16([1:100]));
```

Now write a single-precision IEEE floating point value (32-bits) at address 2000A(Hex).

```
IDE_Obj.write('2000A',single(23.5));
```

For the third example, write a 2-D array of integers in row-major format (standard C programming format) at address 131072 (decimal).

```
m1arr = int32([1:10;101:110]);  
IDE_Obj.write(131072,m1arr');
```

See Also

[hex2dec](#) | [read](#)

xmakefilesetup

Purpose Configure your coder product to generate makefiles

Syntax xmakefilesetup

IDEs This function supports the following IDEs:

Description You can configure your coder product to generate and build your software using makefiles. This process can use the software build toolchains, such as compilers and linkers, associated with the preceding list of IDEs. However, the makefile build process does not use the graphical user interface of the IDE directly.

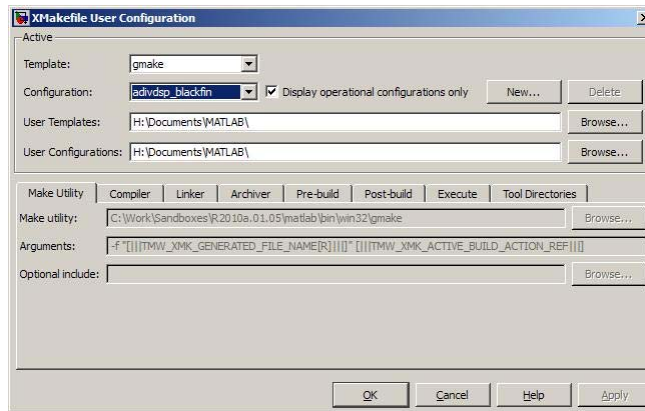
Enter `xmakefilesetup` at the MATLAB command line to configure how to generate makefiles.

Use this function:

- Before you build your software using makefiles for the first time.
- If you change the software build toolchain or processor family.

For more instructions and examples, see “Makefiles for Software Build Tool Chains”.

The `xmakefile` function displays the following dialog box, which prompts you for information about your make utility and software build toolchain.



See Also

“Build format” on page 4-314 | “Build action” on page 4-316

eclipseide

Purpose Create handle object to interact with Eclipse IDE

Syntax
`IDE_Obj = eclipseide`
`IDE_Obj = eclipseide('timeout', period)`

IDEs This function supports the following IDEs:

- Eclipse IDE

Description Before using `eclipseide` for the first time:

- Install the versions of Eclipse IDE and related build tools described in “Installing Third-Party Software for Eclipse™”.
- Use the `eclipseidesetup` function to configure and install a plug-in that enables your coder product to interact with Eclipse IDE.

Use `IDE_Obj = eclipseide` to create an IDE handle object, which you can use to communicate with the Eclipse IDE and processors connected to the Eclipse IDE. After creating the IDE handle object, you can use the methods for the Eclipse IDE.

When you use `eclipseide`, your coder product uses the plug-in to open a session with Eclipse. If Eclipse IDE is not already running, the `eclipseide` function starts the Eclipse IDE. The session connects via the IP port number and uses the workspace you specified previously with `eclipseidesetup`.

When you build a model, the software uses `eclipseide` to create an IDE handle object. In that case, the software gets the name of the IDE handle object from the **IDE link handle name** parameter (default value: `IDE_Obj`) in the configuration parameters for the model.

To assign a timeout period to the handle object, enter the following command:

```
IDE_Obj = eclipseide('timeout', period)
```

For *period*, enter the number of seconds that the handle object waits for processor operations (such as load) to complete. Operations that

exceed the timeout period generate timeout errors. The default period is 10 seconds.

Examples

For example, to create an object handle with a 20-second timeout period, enter:

```
>> IDE_Obj = eclipseide('timeout',20)
Starting Eclipse(TM) IDE...
```

ECLIPSEIDE Object:

```
Default timeout   : 20.00 secs
Eclipse folder    : C:\eclipse3.4\eclipse
Eclipse workspace: C:\WINNT\Profiles\rdlugyhe\workspace
Port number       : 5555
Processor site    : local
```

See Also

`eclipseidesetup`

eclipseidesetup

Purpose	Configure your coder product to interact with Eclipse IDE
Syntax	<code>eclipseidesetup</code>
IDEs	This function supports the following IDEs: <ul style="list-style-type: none">• Eclipse IDE
Description	<p>Before using <code>eclipseidesetup</code> for the first time, install the versions of Eclipse IDE and related build tools described in “Installing Third-Party Software for Eclipse”.</p> <p>To avoid potential build errors later on, close Eclipse IDE before you run <code>eclipseidesetup</code>. For more information, see Build Errors.</p> <p>Use <code>eclipseidesetup</code> at the MATLAB command line to set up your coder product to interact with Eclipse IDE. This action displays a dialog box which you use to configure and add a plugin to the Eclipse IDE. For detailed instructions and examples, see “Configuring Your MathWorks® Software to Work with Eclipse”.</p> <p>When to use <code>eclipseidesetup</code>:</p> <ul style="list-style-type: none">• After you install or reinstall the Eclipse IDE.• Before you use the <code>eclipseide</code> constructor function to create an IDE handle object for the first time.
See Also	<code>eclipseide</code>

Purpose	Working folder used by Eclipse
Syntax	<code>wd= IDE_Obj.pwd</code>
IDEs	This function supports the following IDEs: <ul style="list-style-type: none">• Eclipse IDE
Description	Use <code>wd= IDE_Obj.pwd</code> to get the working folder of the Eclipse IDE. This value is the same as the Eclipse IDE workspace folder.
Examples	To get the Eclipse IDE working folder: <pre>IDE_Obj = eclipseide; wd = IDE_Obj.pwd wd = C:\WINNT\Profiles\rdlugyhe\workspace</pre>
See Also	<code>dir</code>

pwd

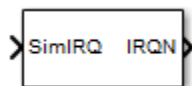
Blocks — Alphabetical List

Async Interrupt

Purpose Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that are to execute downstream subsystems or Task Sync blocks

Library Asynchronous / Interrupt Templates

Description For each specified VxWorks® VME interrupt level, 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

You can use the block for simulation and code generation.

Parameters **VME interrupt number(s)**
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)
An array of unique interrupt vector offset numbers corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field. The Stateflow software passes the offsets to the VxWorks call `intConnect(INUM_TO_IVEC(offset), ...)`.

Simulink task priority(s)

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 you specify for **VME interrupt number(s)**.

The **Simulink task priority** values are required to generate a rate transition code (see “Rate Transitions and Asynchronous Blocks” in the Simulink Coder documentation). Simulink task priority values are also required to maintain absolute time integrity when the asynchronous task needs to 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

The value 1 or 0. 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 VxWorks. To lock out interrupts during the execution of an ISR, set the preemption flag to 0. This causes generation of `intLock()` and `intUnlock()` calls at the beginning and end of the ISR code. Use interrupt locking carefully, as it increases the system’s interrupt response time for interrupts at the `intLockLevelSet()` level and below. Specify an array of flags corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field.

Async Interrupt

Note The number of elements in the arrays specifying **VME interrupt vector offset(s)** and **Simulink task priority** must match the number of elements in the **VME interrupt number(s)** array.

Manage own timer

If checked, the ISR generated by the Async Interrupt block manages its own timer by reading absolute time from the hardware timer. Specify the size of the hardware timer with the **Timer size** option.

Timer resolution (seconds)

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 VxWorks kernel by using the `tickGet` call. The **Timer resolution** field determines the resolution of these counters. The default resolution is 1/60 second. The `tickGet` resolution for your board support package (BSP) might be different. You should determine the `tickGet` resolution for your BSP and enter it in the **Timer resolution** field.

If you are targeting VxWorks, you can obtain better timer resolution by replacing the `tickGet` call and accessing a hardware timer by using your BSP instead. If you are targeting an RTOS other than VxWorks, you should replace the `tickGet` call with an equivalent call to the target RTOS, or generate code to read the timer register on the target hardware. See “Use Timers in Asynchronous Tasks” and “Async Interrupt Block Implementation” in the Simulink Coder documentation for more information.

Timer size

The number of bits to be used to store the clock tick for a hardware timer. The ISR generated by the Async Interrupt block uses the timer size when you select **Manage own timer**. The size can be

32bits (the default), 16bits, 8bits, or auto. If you select auto, the Simulink Coder software determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. However, when **Timer size** is auto, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, the code generator uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters”. See also “Use Timers in Asynchronous Tasks”.

Enable simulation input

If checked, the Simulink software adds an input port to the Async Interrupt block. This port is for use in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Note Before generating code, consider removing blocks that drive the simulation input to prevent the blocks from contributing to the generated code. Alternatively, you can use the Environment Controller block, as explained in “Dual-Model Approach: Code Generation”. However, if you use the Environment Controller block, be aware that the sample times of driving blocks contribute to the sample times supported in the generated code.

Async Interrupt

Inputs and Outputs

Input

A simulated interrupt source.

Output

Control signal for a

- Function-call subsystem
- Task Sync block
- Stateflow chart configured for a function call input event

Assumptions and Limitations

- The block supports VME interrupts 1 through 7.
- The block requires a VxWorks Board Support Package (BSP) that supports the following 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. As a general rule, it is best to keep ISRs as short as possible. Connect only function-call subsystems that contain a small number of blocks to an Async Interrupt block.

A better solution for large subsystems is to use the Task Sync block to synchronize the execution of the function-call subsystem to a VxWorks 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. VxWorks then schedules and runs the task. See the description of the Task Sync block for more information.

See Also

Task Sync

“Handle Asynchronous Events” in the Simulink Coder documentation

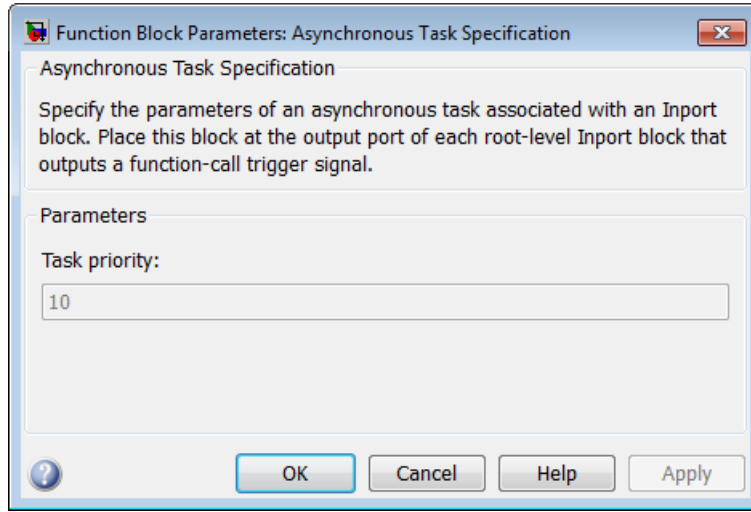
Asynchronous Task Specification

Purpose	Allow for parameter specifications for asynchronous tasks associated with root-level Inport blocks that output a function-call trigger
Library	Asynchronous
Description	<p>The Asynchronous Task Specification block, in combination with a root-level Inport block, allows for an asynchronous function-call input to a model reference.</p> <p>To implement this feature, place this block at the output port of each root-level Inport block that outputs a function-call trigger. On the Signal Attributes pane of the Inport block, select Output function call to specify that the Inport block accepts function-call signals. Then use the Asynchronous Task Specification blocks to specify the asynchronous task parameters associated with the respective Inport blocks.</p>
Data Type Support	This specification does not apply to the Asynchronous Task Specification block; the block accepts only function-call signals.

Asynchronous Task Specification

Parameters and Dialog Box

The **Function Block Parameters** dialog box of the Asynchronous Task Specification block appears as follows:



Asynchronous Task Specification

Task priority

Specifies the priority of the asynchronous task calling the destination function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

Settings

Default: 10

- You can enter an integer or [].
- If you specify an integer for an Asynchronous Task Specification block that resides in a referenced model, the priority of the initiator in the top model must match the priority of that block.
- If you specify [] for an Asynchronous Task Specification block that resides in a referenced model, the priority of the initiator in the top model does not have to match the priority of that block. For this case, the rate transition algorithm is conservative (not optimized), assuming that the priority is unknown but static.

Command-Line Information

This block has only one parameter.

Parameter: TaskPriority

Value: integer

Configuration Parameters Settings

To create an asynchronous model reference containing a Function-Call and an Asynchronous Task Specification block, you must follow the procedure outlined in “Convert an Asynchronous Subsystem into a Model Reference”. One of the steps requires that you make several changes to configuration parameters.

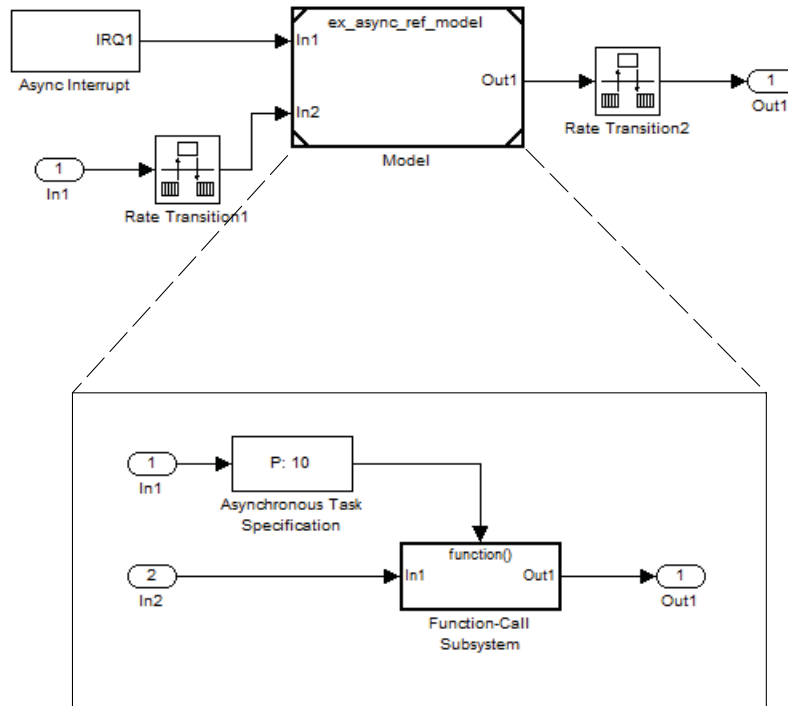
Additional configuration parameters that require attention are the solver **Type** and the **Fixed step size (fundamental sample time)** on the Solver pane. Both the top model and the referenced model must use a fixed-step solver. Moreover, the referenced model must have a fundamental sample time that is an integer multiple of the fundamental sample time of the top model.

Asynchronous Task Specification

Examples

Asynchronous Function-Call Input to Model

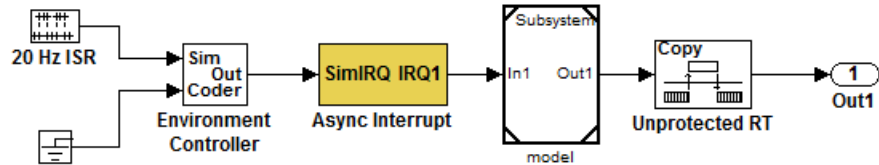
This root-level model uses the Inport block with the Asynchronous Task Specification block to allow a function-call input signal to a model reference. The priority is set to 10.



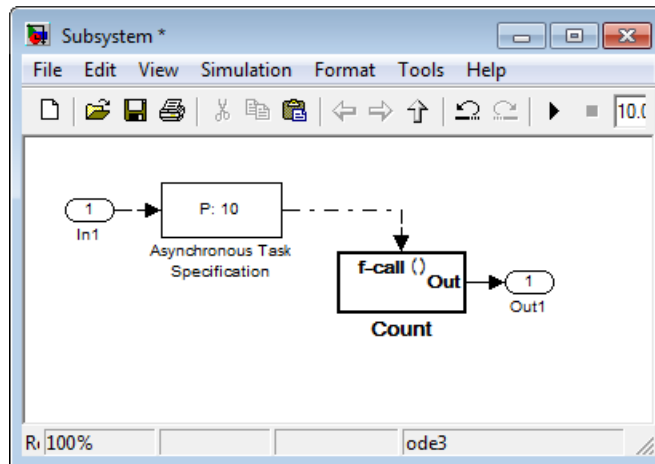
The Asynchronous Task Specification block must immediately follow the Inport block. Also, a branch cannot emanate from the signal connecting the Inport block to the Asynchronous Task Specification block.

Asynchronous Task Specification

Setting Priorities

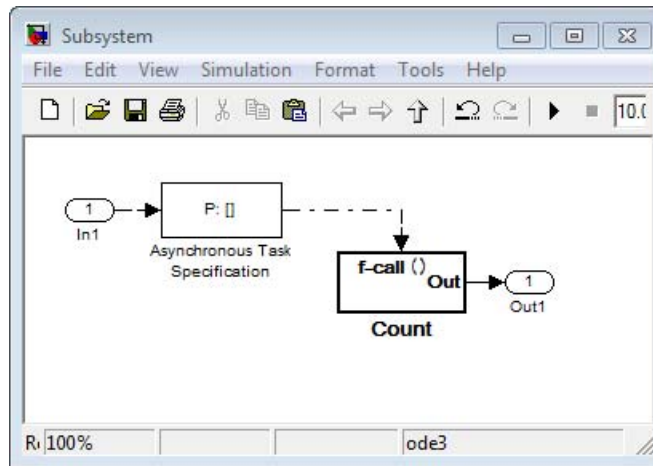


For this model, if the Asynchronous Task Specification block is set to the default value of 10, the Async Interrupt block must also have a priority of 10.



Whereas, if the priority of the Asynchronous Task Specification block is set to the empty matrix, [], the priority of the Async Interrupt can be a value other than 10.

Asynchronous Task Specification



Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	N/A
Dimensionalized	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Function-Call Subsystem block
"Handle Asynchronous Events"
"Model Reference"
Inport block

Byte Pack

Purpose Convert input signals to uint8 vector

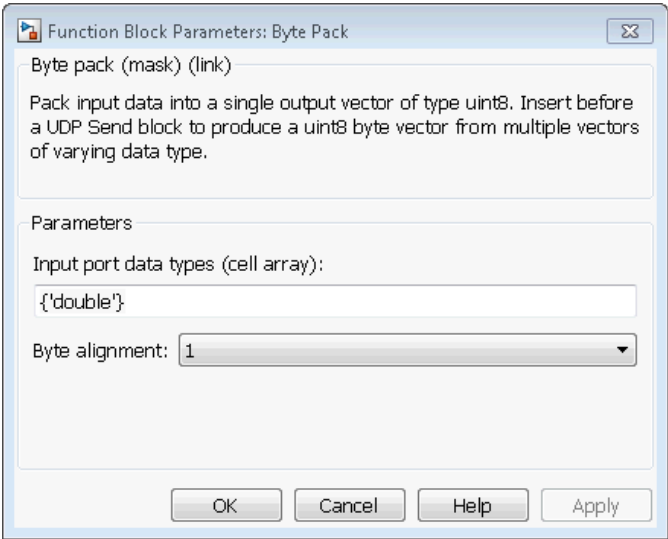
Library Embedded Coder/ Embedded Targets/ Host Communication
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments™
C6000/ Target Communication
Simulink Coder / Desktop Targets/ Host Communication



Description

Using the input port, the block converts data of one or more data types into a single uint8 vector for output. With the options available, you specify the input data types and the alignment of the data in the output vector. Because UDP messages are in uint8 data format, use this block before a UDP Send block to format the data for transmission using the UDP protocol.

Dialog Box



Input port data types (cell array)

Specify the data types for the different signals as part of the parameters. The block supports all Simulink data types except characters. Enter the data types as Simulink types in the cell array, such as 'double' or 'int32'. The order of the data type entries in the cell array must match the order in which the data arrives at the block input. This block determines the signal sizes automatically. The block has at least one input port and only one output port.

Byte alignment

This option specifies how to align the data types to form the uint8 output vector. Select one of the values in bytes from the list.

Alignment can occur on 1, 2, 4, or 8-byte boundaries depending on the value you choose. The value defaults to 1. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithm is that each element in the output vector begins on a byte boundary specified by the

Byte Pack

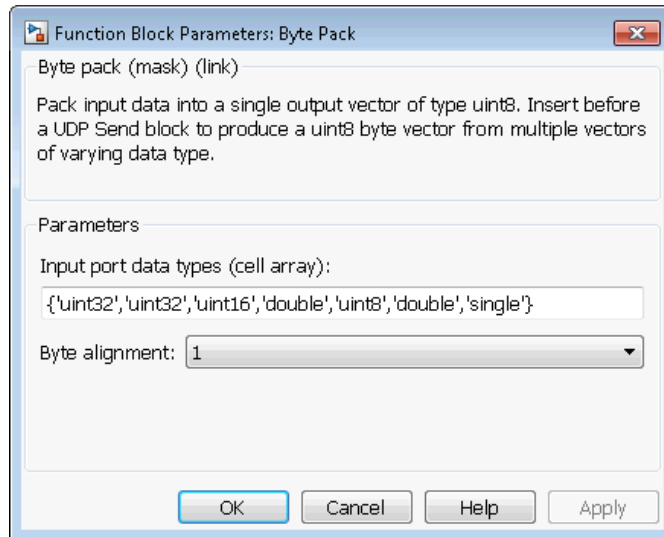
alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

Selecting 1 for **Byte alignment** provides the tightest packing, without holes between data types in the various combinations of data types and signals.

Sometimes, you can have multiple data types of varying lengths. In such cases, specifying a 2-byte alignment can produce 1-byte gaps between uint8 or int8 values and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time. You can specify data alignment options with data types.

Example

Use a cell array to enter input data types in the **Input port data types** parameter. The order of the data types you enter must match the order of the data types at the block input.



In the cell array, you provide the order in which the block expects to receive data—uint32, uint32, uint16, double, uint8, double, and

`single`. With this information, the block automatically provides the number of block inputs.

Byte alignment equal to 2 specifies that each new value begins 2 bytes from the previous data boundary.

The example shows the following data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

When the signals are scalar values (not matrices or vectors in this example), the first signal value in the vector starts at 0 bytes. Then, the second signal value starts at 2 bytes, and the third at 4 bytes. Next, the fourth signal value follows at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. As the example shows, the packing algorithm leaves a 1-byte gap between the `uint8` data value and the `double` value.

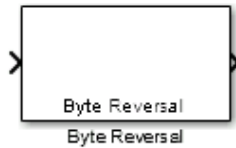
See Also

Byte Reversal, Byte Unpack

Byte Reversal

Purpose Reverse order of bytes in input word

Library Embedded Coder/ Embedded Targets/ Host Communication
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments
C6000/ Target Communication
Simulink Coder / Desktop Targets/ Host Communication

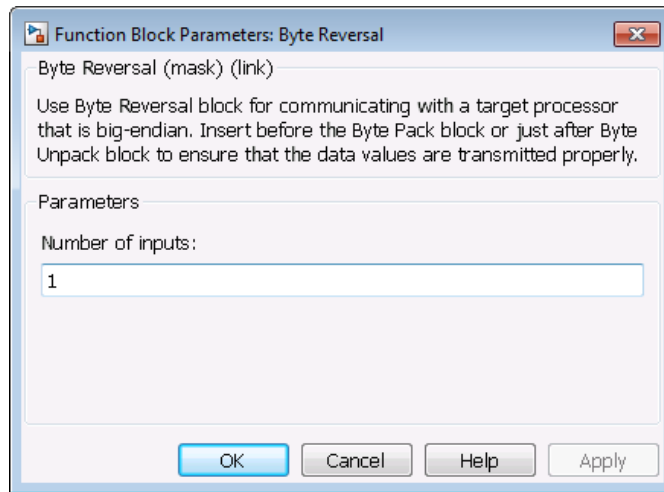


Description

Byte reversal changes the order of the bytes in data you input to the block. Use this block when your process communicates between targets that use different endianness, such as between Intel® processors that are little endian and others that are big endian. Texas Instruments processors are little-endian by default.

To exchange data with a processor that has different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

Dialog Box



Number of inputs

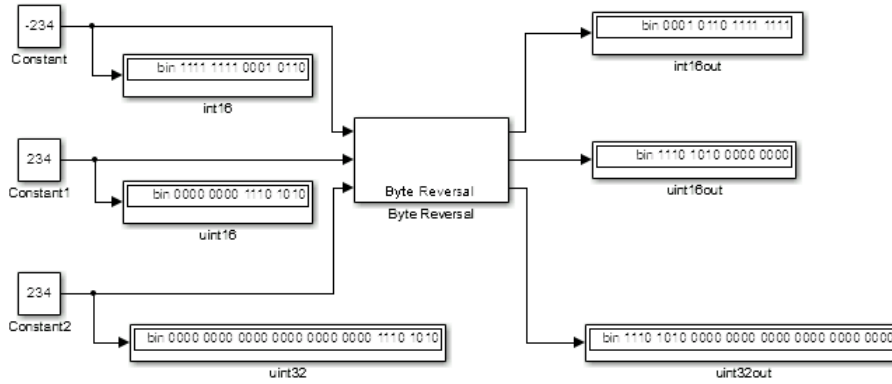
Specify the number of block inputs. The number of block inputs adjusts automatically to match value so the number of outputs equals the number of inputs.

When you use more than one input port, each input port maps to the matching output port. Data entering input port 1 leaves through output port 1, and so on.

Reversing the bytes does not change the data type. Input and output retain matching data type.

The following model shows byte reversal in use. In this figure, the input and output ports match for each path.

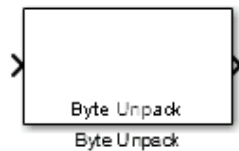
Byte Reversal



See Also Byte Pack, Byte Unpack

Purpose Unpack UDP uint8 input vector into Simulink data type values

Library Embedded Coder/ Embedded Targets/ Host Communication
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments
C6000/ Target Communication
Simulink Coder / Desktop Targets/ Host Communication

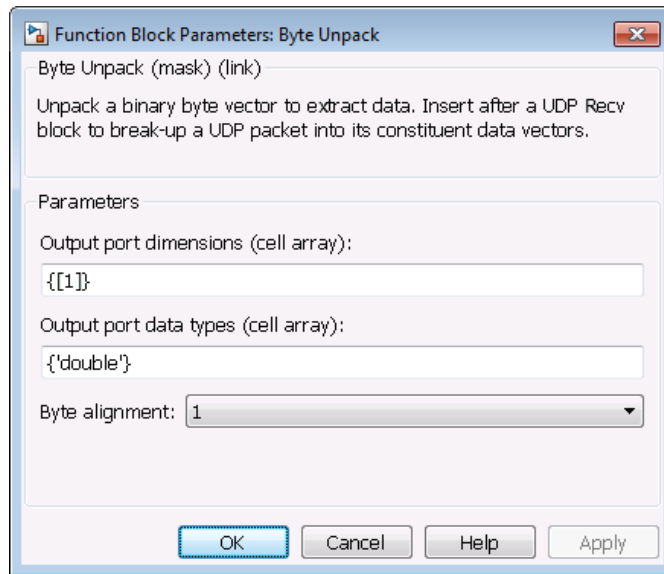


Description

Byte Unpack is the inverse of the Byte Pack block. It takes a UDP message from a UDP receive block as a `uint8` vector, and outputs Simulink data types in various sizes depending on the input vector.

The block supports all Simulink data types.

Byte Unpack



Dialog Box

Output port dimensions (cell array)

Containing a cell array, each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding Byte Pack block in the model. Entering one value means that the block applies that dimension to all data types.

Output port data types (cell array)

Specify the data types for the different input signals to the Pack block. The block supports all Simulink data types—`single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`, and `Boolean`. The entry here is the same as the Input port data types parameter in the Byte Pack block in the model. You can enter one data type and the block applies that type to all output ports.

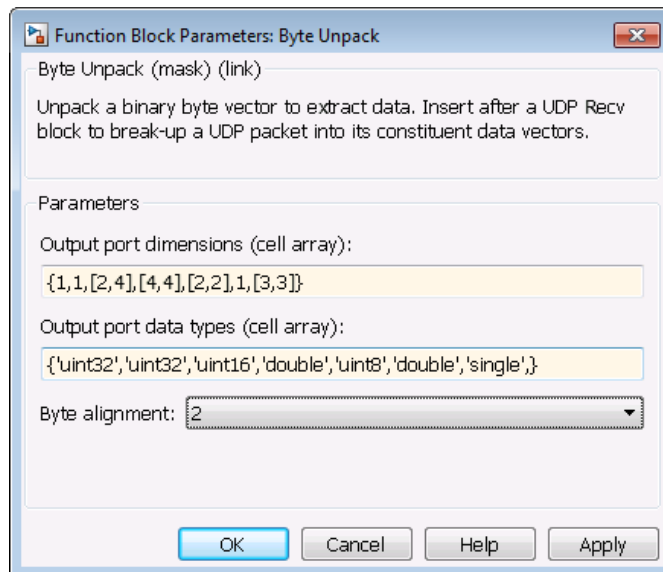
Byte Alignment

This option specifies how to align the data types to form the input uint8 vector. Match this setting with the corresponding Byte Pack block alignment value of 1, 2, 4, or 8 bytes.

Example

This figure shows the Byte Unpack block that corresponds to the example in the Byte Pack example. The **Output port data types (cell array)** entry shown is the same as the **Input port data types (cell array)** entry in the Byte Pack block

```
{'uint32','uint32','uint16','double','uint8','double','single'}.
```



In addition, the **Byte alignment** setting matches as well. **Output port dimensions (cell array)** now includes scalar values and matrices to show how to enter nonscalar values. The example for the Byte Pack block assumed only scalar inputs.

See Also

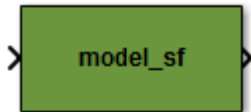
Byte Pack, Byte Reversal

Generated S-Function

Purpose Represent model or subsystem as generated S-function code

Library S-Function Target

Description



An instance of the Generated S-Function block represents code the Simulink Coder software generates from its S-function target for a model or subsystem. For example, you extract a subsystem from a model and build a Generated S-Function block from it, using the S-function target. This mechanism can be useful for

- Converting models and subsystems to application components
- Reusing models and subsystems
- Optimizing simulation — often, an S-function simulates more efficiently than the original model

For details on how to create a Generated S-Function block from a subsystem, see “Create S-Function Blocks from a Subsystem” in the Simulink Coder documentation.

- 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”.
 - You must set the solver parameters of the Generated S-Function block to be the same as those of the original model or subsystem. The generated S-function code will operate identically to the

original subsystem (see Choice of Solver Type in the Simulink Coder documentation for an exception to this rule).

Parameters

Generated S-function name (`model_sf`)

The name of the generated S-function. The Simulink Coder software derives the name by appending `_sf` to the name of the model or subsystem from which the block is generated.

Show module list

If checked, displays modules generated for the S-function.

See Also

“Create S-Function Blocks from a Subsystem” in the Simulink Coder documentation

Linux Audio Capture

Purpose

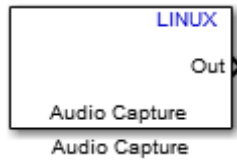
Capture ALSA audio from sound card and output data

Library

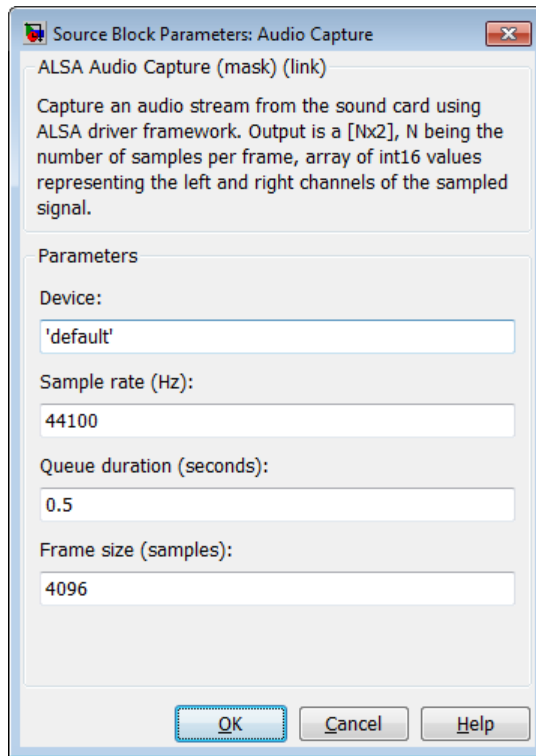
Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux®

Simulink Coder / Desktop Targets/ Operating Systems/ Linux

Description



This block uses the ALSA driver framework to capture an audio stream from a sound card. It outputs the left and right channels of the signal as an $[N \times 2]$ frame of int16 values. N is the number of samples per frame.



Dialog

Device

Use the default ALSA device, or enter the name of a specific audio output device.

Entering 'default' selects the ALSA device specified by an ALSA configuration file on your target Linux system.

One of the following ALSA configuration files defines the default device:

- `/etc/asound.conf`, which defines system-wide options for all users

Linux Audio Capture

- `~/.asoundrc`, which overrides `/etc/asound.conf` for the current user

The entry that specifies the default device looks similar to this example:

```
pcm.!default {
    type hw
    card 0
    device 2
}
```

To enter the name of an alternate audio input device, review the `/proc/asound/cards` file on your target Linux system. For example, if `/proc/asound/cards` contained the following entries, you could set the value of **Device** to 'AudioPCI' :

```
$ cat /proc/asound/cards

0 [Dummy      ]: Dummy - Dummy
                  Dummy 1

1 [VirMIDI    ]: VirMIDI - VirMIDI
                  Virtual MIDI Card 1

2 [AudioPCI   ]: ENS1371 - Ensoniq AudioPCI
                  Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

The default value for **Device** is 'default'.

Sample rate (Hz)

Enter a value that matches the sample rate of the ALSA audio output.

By default, the sample rate of the ALSA output equals the output of the audio capture device. In this case, enter the sample rate of the audio capture device.

The `/etc/asound.conf` and `~/.asoundrc` files can configure ALSA to downsample the signal from the audio capture device. In this case, enter the downsample rate specified by the configuration files. For example, if one of the configuration files contained the following entry, you would set the value of **Sample rate (Hz)** to 16000 :

```
pcm_slave.sl3 {
    pcm ens1371
    format S16_LE
    channels 1
    rate 16000
}
pcm.complex_convert {
    type plug
    slave sl3
}
```

The default value for **Sample rate (Hz)** is 44100 Hz (44.1 kHz). The maximum rate equals the sampling rate of the audio capture device.

Queue duration (seconds)

Set the duration of the queue in seconds. This queue provides a software-based frame buffer between the ALSA output and the Linux Audio Capture block. The queue prevents dropped data caused by temporary mismatches in the rate of data arriving and leaving the queue. Higher values can handle more significant mismatches, but such values also increase signal latency and memory usage.

The default value for **Queue duration (seconds)** is 0.5 seconds.

Frame size (samples)

Set the number of samples per frame in the output this block sends to your model. The default value for this parameter is 4096 samples.

Linux Audio Capture

References <http://www.alsa-project.org>

See Also <http://www.alsa-project.org>
Linux Audio Playback
Linux Task

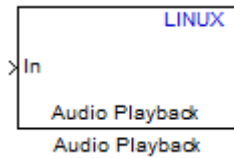
Purpose

Send audio data stream to ALSA audio device output

Library

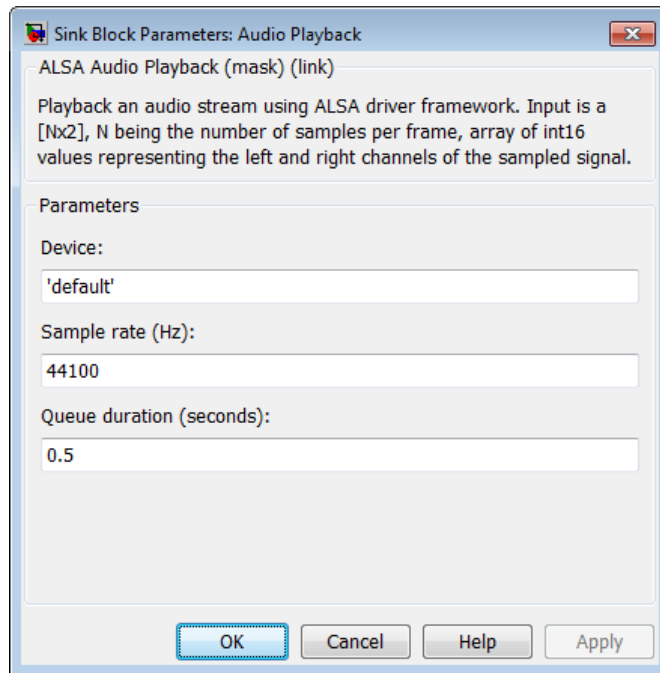
Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux (linuxlib)

Simulink Coder / Desktop Targets/ Operating Systems/ Linux

**Description**

This block takes a stream of audio data and sends it to the output jack of an ALSA audio device. The block input, **In**, takes the left and right channels of data as an [Nx2] frame of int16 values. N is the number of samples per frame.

Linux Audio Playback



Dialog

Device

Use the default ALSA device, or enter the name of a specific audio device.

Entering 'default' selects the ALSA device specified by an ALSA configuration file on your target Linux system.

One of the following ALSA configuration files defines the default device:

- `/etc/asound.conf`, which defines system-wide options for all users
- `~/.asoundrc`, which overrides `/etc/asound.conf` for the current user

The entry that specifies the default device looks like this hypothetical example:

```
pcm.!default {
    type hw
    card 0
    device 2
}
```

To enter the name of an alternate audio device, consult the `/proc/asound/cards` file on your target Linux system. For example, if `/proc/asound/cards` contained the following hypothetical entries, you could set the value of **Device** to 'AudioPCI' :

```
$ cat /proc/asound/cards

0 [Dummy      ]: Dummy - Dummy
                   Dummy 1

1 [VirMIDI    ]: VirMIDI - VirMIDI
                   Virtual MIDI Card 1

2 [AudioPCI   ]: ENS1371 - Ensoniq AudioPCI
                   Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

The default value for **Device** is 'default'.

Sample rate (Hz)

Enter a value that matches the sample rate of the ALSA audio output.

By default, the sample rate of the ALSA output is the same as the output of the audio capture device. In this case, enter the sample rate of the audio capture device.

The `/etc/asound.conf` and `~/.asoundrc` files can configure ALSA to downsample the signal from the audio capture device. In

Linux Audio Playback

this case, enter the downsample rate specified by the configuration files. For example, if one of the configuration files contained the following hypothetical entry, you would set the value of **Sample rate (Hz)** to 16000 :

```
pcm_slave.s13 {
    pcm ens1371
    format S16_LE
    channels 1
    rate 16000
}
pcm.complex_convert {
    type plug
    slave s13
}
```

The default value for **Sample rate (Hz)** is 44100 Hz (44.1 kHz). The maximum rate is the sampling rate of the audio capture device.

Queue duration (seconds)

Set the duration of the queue in seconds. This queue provides a software-based frame buffer between the ALSA audio device and this block. The queue prevents dropped data caused by temporary mismatches in the rate of data arriving and leaving the queue. Higher values can handle more significant mismatches, but increase signal latency and memory usage.

The default value for **Queue duration (seconds)** is 0.5 seconds.

See Also

<http://www.alsa-project.org>

Linux Audio Capture

Linux Task

Purpose

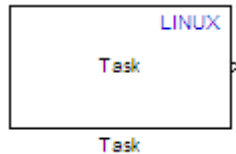
Spawn task function as separate Linux thread

Library

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

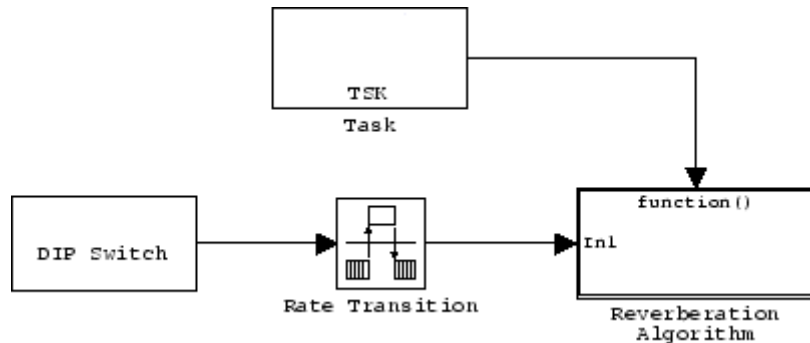
Embedded Coder Support Package for Xilinx® Zynq®-7000 Platform

Simulink Coder / Desktop Targets/ Operating Systems/ Linux



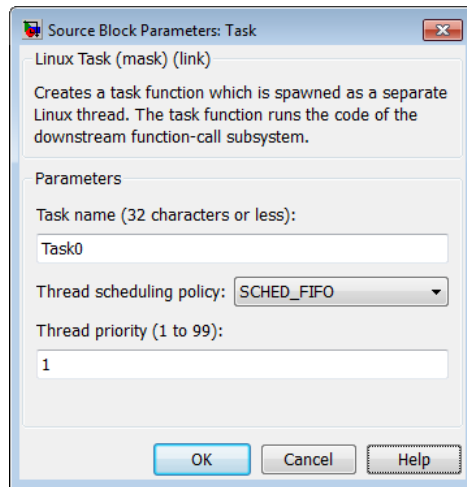
Description

Use this block to create a task function that spawns as a separate Linux thread. The task function runs the code of the downstream function-call subsystem. For example:



In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_ert.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

Dialog



Task name

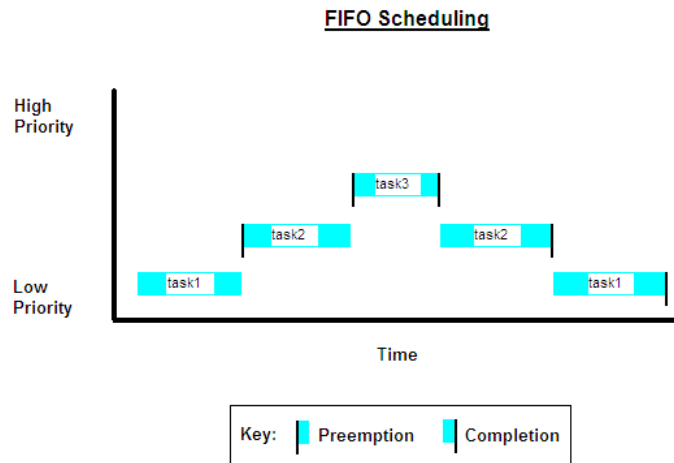
Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

Thread scheduling policy

Select the scheduling policy that applies to this thread. You can choose from the following options:

- **SCHED_FIFO** enables a First In, First Out scheduling algorithm that executes real-time processes without time slicing. With FIFO scheduling, a higher-priority process preempts a lower-priority process. The lower-priority process remains at the top of the list for its priority and resumes execution when the scheduler blocks all higher-priority processes.

For example, in the following image, task2 preempts task1. Then task3 preempts task2. When task3 completes, task2 resumes. When task2 completes, task1 resumes.



Selecting `SCHED_FIFO`, displays the **Thread priority** parameter, which you can set to a value from 1 to 99.

- `SCHED_OTHER` enables the default Linux time-sharing scheduling algorithm. You can use this scheduling for all processes except those requiring special static priority real-time mechanisms. With this algorithm, the scheduler chooses processes based on their dynamic priority within the static priority 0 list. Each time the process is ready to run and the scheduler denies it, the operating system increases that process's dynamic priority. Such prioritization helps the scheduler serve the `SCHED_OTHER` processes.

Selecting `SCHED_OTHER`, hides the **Thread priority** parameter, and sets the thread priority to 0.

Thread priority (1 to 99)

When you set **Thread scheduling policy** to `SCHED_FIFO`, you can set the priority of the thread from 1 to 99 (low-to-high).

Higher-priority tasks can preempt lower-priority tasks.

See Also

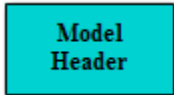
Linux Audio Capture

Linux Audio Playback

Purpose Specify custom header code

Library Custom Code

Description The Model Header block adds user-specified custom code to the *model.h* file that the code generator creates for the model that contains the block.



Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

Top of Model Header

Code to be added near the top of the generated model header file, in a `user code (top of header file)` section.

Bottom of Model Header

Code to be added at the bottom of the generated model header file, in a `user code (bottom of header file)` section.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

See Also

Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

Model Source

Purpose	Specify custom source code
Library	Custom Code
Description	The Model Source block adds user-specified custom code to the <i>model.c</i> or <i>model.cpp</i> file that the code generator creates for the model that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters	Top of Model Source Code to be added near the top of the generated model source file, in a user code (top of source file) section.
	Bottom of Model Source Code to be added at the bottom of the generated model source file, in a user code (bottom of source file) section.

Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

Purpose	Handle transfer of data between blocks operating at different rates and maintain data integrity
Library	VxWorks (vxlib1)
Description	The Protected RT block is a Rate Transition block that is preconfigured to maintain data integrity during data transfers. For more information, see Rate Transition in the Simulink Reference.

System Derivatives

Purpose	Specify custom system derivative code
Library	Custom Code
Description	The System Derivatives block adds user-specified custom code to the declaration, execution, and exit code sections of the <code>SystemDerivatives</code> function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters	System Derivatives Function Declaration Code Code to be added to the declaration section of the generated <code>SystemDerivatives</code> function.
	System Derivatives Function Execution Code Code to be added to the execution section of the generated <code>SystemDerivatives</code> function.
	System Derivatives Function Exit Code Code to be added to the exit section of the generated <code>SystemDerivatives</code> function.

Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, Model Source, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

Purpose	Specify custom system disable code
Library	Custom Code
Description	The System Disable block adds user-specified custom code to the declaration, execution, and exit code sections of the <code>SystemDisable</code> function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters	System Disable Function Declaration Code Code to be added to the declaration section of the generated <code>SystemDisable</code> function.
	System Disable Function Execution Code Code to be added to the execution section of the generated <code>SystemDisable</code> function.
	System Disable Function Exit Code Code to be added to the exit section of the generated <code>SystemDisable</code> function.

Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, Model Source, System Derivatives, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

System Enable

Purpose Specify custom system enable code

Library Custom Code

Description The System Enable block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemEnable` function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Enable Function Declaration Code
Code to be added to the declaration section of the generated `SystemEnable` function.

System Enable Function Execution Code
Code to be added to the execution section of the generated `SystemEnable` function.

System Enable Function Exit Code
Code to be added to the exit section of the generated `SystemEnable` function.

Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, Model Source, System Derivatives, System Disable, System Initialize, System Outputs, System Start, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

Purpose	Specify custom system initialization code
Library	Custom Code
Description	The System Initialize block adds user-specified custom code to the declaration, execution, and exit code sections of the <code>SystemInitialize</code> function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters	System Initialize Function Declaration Code Code to be added to the declaration section of the generated <code>SystemInitialize</code> function.
	System Initialize Function Execution Code Code to be added to the execution section of the generated <code>SystemInitialize</code> function.
	System Initialize Function Exit Code Code to be added to the exit section of the generated <code>SystemInitialize</code> function.

Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, Model Source, System Derivatives, System Disable, System Enable, System Outputs, System Start, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

System Outputs

Purpose Specify custom system outputs code

Library Custom Code

Description The System Outputs block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemOutputs` function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters **System Outputs Function Declaration Code**
Code to be added to the declaration section of the generated `SystemOutputs` function.

System Outputs Function Execution Code
Code to be added to the execution section of the generated `SystemOutputs` function.

System Outputs Function Exit Code
Code to be added to the exit section of the generated `SystemOutputs` function.

Example See “Embed Custom Code Directly Into `MdlStart` Function”.

See Also Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Start, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

Purpose	Specify custom system startup code
Library	Custom Code
Description	The System Start block adds user-specified custom code to the declaration, execution, and exit code sections of the <code>SystemStart</code> function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters	System Start Function Declaration Code Code to be added to the declaration section of the generated <code>SystemStart</code> function.
	System Start Function Execution Code Code to be added to the execution section of the generated <code>SystemStart</code> function.
	System Start Function Exit Code Code to be added to the exit section of the generated <code>SystemStart</code> function.

Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Terminate, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

System Terminate

Purpose Specify custom system termination code

Library Custom Code

Description The System Terminate block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemTerminate` function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters **System Terminate Function Declaration Code**
Code to be added to the declaration section of the generated `SystemTerminate` function.

System Terminate Function Execution Code
Code to be added to the execution section of the generated `SystemTerminate` function.

System Terminate Function Exit Code
Code to be added to the exit section of the generated `SystemTerminate` function.

Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Update
“Insert Custom Code Blocks” in the Simulink Coder documentation

Purpose	Specify custom system update code
Library	Custom Code
Description	The System Update block adds user-specified custom code to the declaration, execution, and exit code sections of the <code>SystemUpdate</code> function that the code generator creates for the model or subsystem that contains the block.

Note If you include this block in a referenced model (model referenced by a Model block), the Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters	System Update Function Declaration Code Code to be added to the declaration section of the generated <code>SystemUpdate</code> function.
	System Update Function Execution Code Code to be added to the execution section of the generated <code>SystemUpdate</code> function.
	System Update Function Exit Code Code to be added to the exit section of the generated <code>SystemUpdate</code> function.

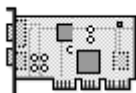
Example See “Embed Custom Code Directly Into MdlStart Function”.

See Also Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate
“Insert Custom Code Blocks” in the Simulink Coder documentation

Target Preferences (Removed)

Purpose Configure model for specific IDE, tool chain, board, and processor

Library Simulink Coder / Desktop Targets
Embedded Coder/ Embedded Targets



Target Preferences

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 Target Hardware Resources tab, located in the Configuration Parameters dialog. For more information, see:

- “Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog”
- “Configure Target Hardware Resources”
- “Code Generation: Coder Target Pane” on page 4-309

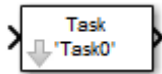
Purpose

Spawn VxWorks task to run code of downstream function-call subsystem or Stateflow chart

Library

Asynchronous / Interrupt Templates

Description



The Task Sync block spawns a 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 might 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 performs the following functions:

- Uses the 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, using `semTake`. The first call to `semTake` specifies `NO_WAIT`. This allows the task to determine whether a second `semGive` has occurred prior to the completion of the function-call subsystem or chart. This would indicate that the interrupt rate is too fast or the task priority is too low.
- Generates synchronization code (for example, `semGive()`). This code allows the spawned task to run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. This is accomplished through the connection between the Async Interrupt and Task Sync blocks, which triggers execution of the Task Sync block within an ISR.
- Supplies absolute time if blocks in the downstream algorithmic code require it. The time is supplied either by the timer maintained by

Task Sync

the Async Interrupt block, or by 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 should 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 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 is driven by 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.

Parameters

Task name (10 characters or less)

The first argument passed to the VxWorks `taskSpawn` system call. 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)

The VxWorks task priority to be assigned to the function-call subsystem task when spawned. 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 to which the task's stack can grow. The stack size is allocated when VxWorks spawns the task. Choose a stack size based on the number of local variables in the task. You should

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

If not checked (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 checked,

- 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 “Use Timers in Asynchronous Tasks” in the Simulink Coder documentation). The timer value is read at the time the asynchronous interrupt is serviced, and data transfers to blocks called by the Task Sync block and execute within the task associated with the Async Interrupt block. Therefore, data transfers are synchronized with the caller.

Timer resolution (seconds)

The resolution of the block’s timer in seconds. This option appears only if **Synchronize the data transfer of this task with the caller task** is not checked. By default, the block gets the timer value by calling the VxWorks `tickGet` function. The default resolution is 1/60 second. The `tickGet` resolution for your BSP might be different. You should determine the `tickGet` resolution for your BSP and enter it in the **Timer resolution** field.

Task Sync

Timer size

The number of bits to be used 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 Simulink Coder software determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. However, when **Timer size** is auto, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, it uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters”. See also “Use Timers in Asynchronous Tasks”.

Inputs and Outputs

Input

A call from an Async Interrupt block.

Output

A call to a function-call subsystem.

See Also

Async Interrupt

“Handle Asynchronous Events” in the Simulink Coder documentation

Purpose

Receive UDP packet

Library

Embedded Coder/ Embedded Targets/ Host Communication
Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux
Embedded Coder Support Package for Wind River® VxWorks RTOS
Embedded Coder Support Package for Xilinx Zynq-7000 Platform
Simulink Coder / Desktop Targets/ Host Communication
Windows (windowplib)

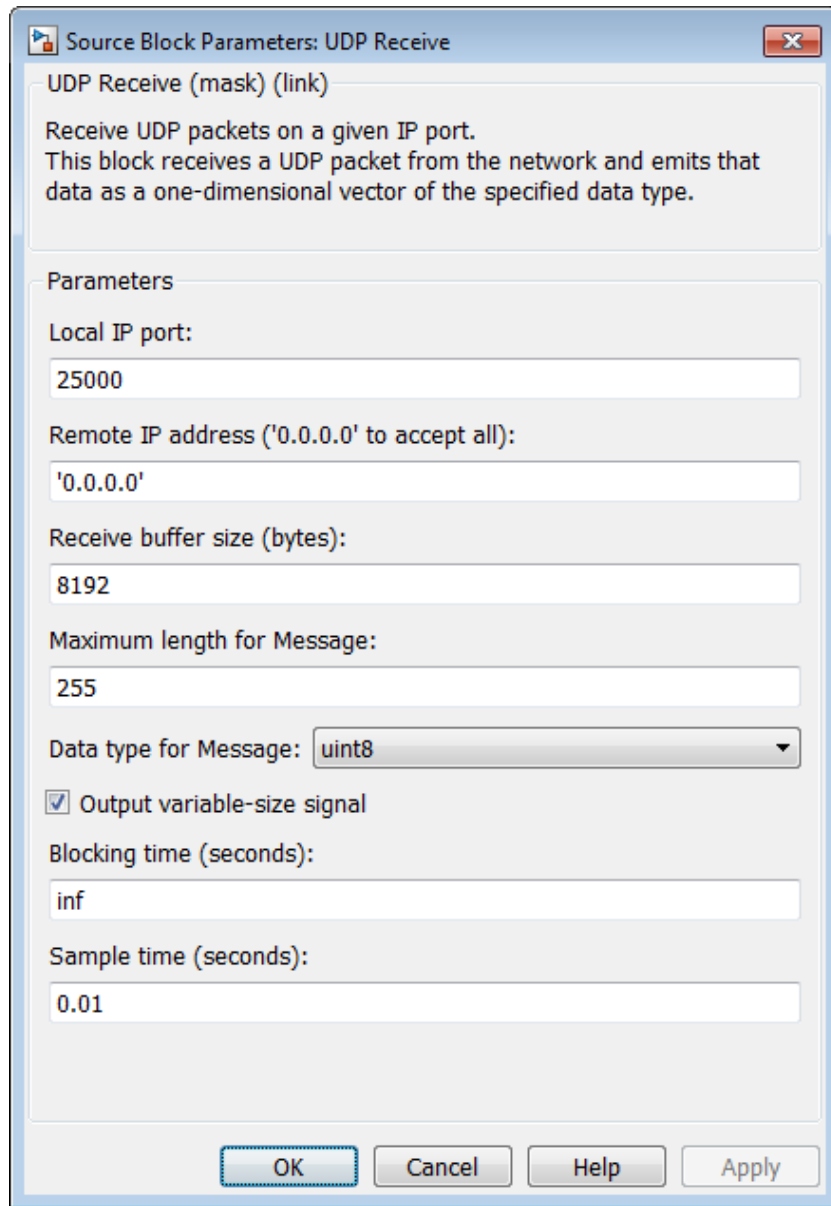
Note If your target system uses Linux or Windows, get the UDP block from `linuxlib` or `windowplib`.

Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block output, emits the contents of a single UDP packet as a data vector.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The `packNGo` function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

UDP Receive



Dialog

Local IP port

Specify the IP port number upon which to receive UDP packets. This value defaults to 25000. The value can range 1–65535.

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Remote IP address ('0.0.0.0' to accept all)

Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from other addresses. To accept packets from any IP address, enter '0.0.0.0'. This value defaults to '0.0.0.0'.

Receive buffer size (bytes)

Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to 8192.

Maximum length for Message

Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of a UDP packet. The system truncates data that exceeds this length. This value defaults to 255.

If you disable **Output variable-size signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

Data type for Message

Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Output variable-size signal

If your model supports signals of varying length, enable the **Output variable-size signal** parameter. This checkbox defaults to selected (enabled). In that case:

UDP Receive

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Data type for Length

Set the data type of the Length output. This option defaults to double.

Blocking time (seconds)

For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note This parameter appears only in the Embedded Coder UDP Receive block.

Sample time (seconds)

Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a

large value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

Output port width

Specify the width of packets the block accepts. When you design the transmit end of the UDP communication channel, you decide the packet width. Set this option to a value as large or larger than a packet you expect to receive.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

UDP receive buffer size (bytes)

Specify the size of the buffer to which the system stores UDP packets. The default size is 8192 bytes. Make the buffer large enough to store UDP packets that come in while your process reads a packet from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Send

UDP Send

Purpose

Send UDP message

Library

Embedded Coder/ Embedded Targets/ Host Communication

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

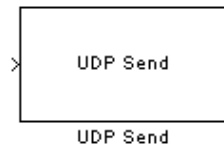
Embedded Coder Support Package for Wind River VxWorks RTOS

Embedded Coder Support Package for Xilinx Zynq-7000 Platform

Simulink Coder / Desktop Targets/ Host Communication

Windows (windowslib)

Note If your target system uses Linux or Windows, get the UDP block from `linuxlib` or `windowslib`.

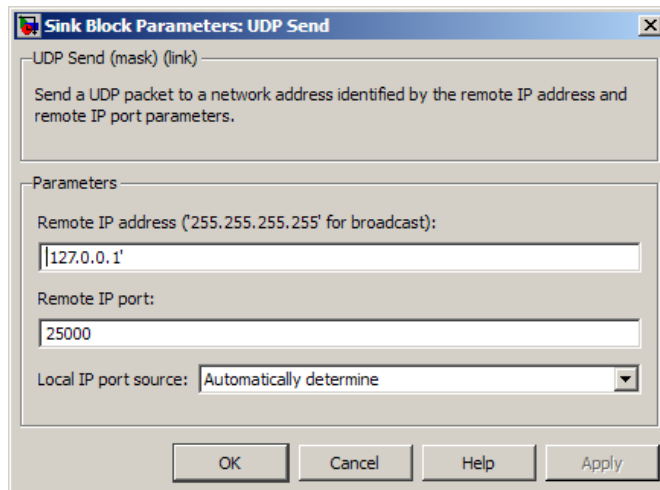


Description

The UDP Send block transmits an input vector as a UDP message over an IP network port.

Note Some Simulink blocks and `.exe` files built from models that contain those blocks require shared libraries, such as `.dll` files on Windows. The UDP Send block requires `networkdevice.dll`. To meet this requirement, open the `packNGo` topic, and follow the example to package the code files for your model. The resulting compressed folder contains the `.dll` files that the model requires, including `networkdevice.dll`. To run this type of `.exe` file outside a MATLAB environment, place the required `.dll` files in the same folder as the `.exe` file, or place them in a folder on the Windows system path.

Dialog Box



IP address ('255.255.255.255' for broadcast)

Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, '255.255.255.255'.

Remote IP port

Specify the port to which the block sends the message. The value defaults to 25000, but the values range from 1–65535.

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

UDP Send

Local IP port source

To let the system automatically assign the port number, select **Assign automatically**. To specify the IP port number using the **Local IP port** parameter, select **Specify**.

Local IP port

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

Sample time

Sample time tells the block how long to wait before polling for new messages.

Note This parameter only appears in a deprecated version of the UDP Send block. Replace the deprecated UDP Send block with a current UDP Send block.

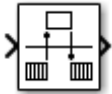
See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Receive

Purpose Handle transfer of data between blocks operating at different rates and maintain determinism

Library VxWorks (vxlib1)

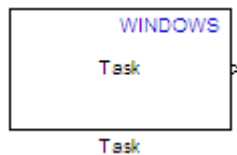
Description The Unprotected RT block is a Rate Transition block that is preconfigured to conduct deterministic data transfers. For more information, see Rate Transition in the Simulink Reference.



Windows Task

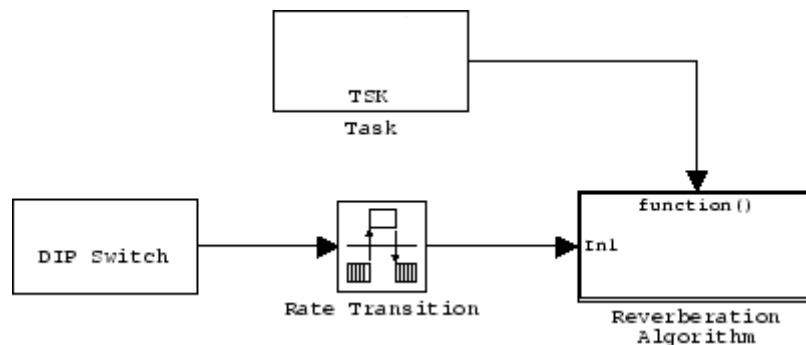
Purpose Spawn task function as separate Windows thread

Library Windows (windowlib)



Description

This block spawns a task function as a separate Windows thread. The task function runs the code of the downstream function-call subsystem. For example:



In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_ert.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

Thread priority in Windows operating systems ranges from 0 to 31 (low-to-high priority). The following two criteria determine the priority of a given thread:

- Priority class
- Priority level within the priority class

The priority classes in Windows are as follows:

- `IDLE_PRIORITY_CLASS`
- `BELOW_NORMAL_PRIORITY_CLASS`
- `NORMAL_PRIORITY_CLASS`
- `ABOVE_NORMAL_PRIORITY_CLASS`
- `HIGH_PRIORITY_CLASS`
- `REALTIME_PRIORITY_CLASS`

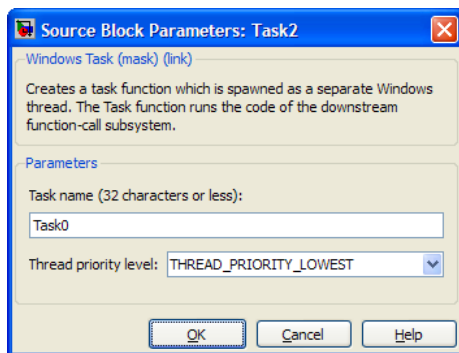
The Windows Task block uses a process priority of `NORMAL_PRIORITY_CLASS`.

In the Windows Task block, you can use the **Thread priority level** parameter specify the following the priority levels within in the `NORMAL_PRIORITY_CLASS`:

- `THREAD_PRIORITY_LOWEST`
- `THREAD_PRIORITY_BELOW_NORMAL`
- `THREAD_PRIORITY_NORMAL`
- `THREAD_PRIORITY_ABOVE_NORMAL`
- `THREAD_PRIORITY_HIGHEST`

Windows Task

Dialog



Task name

Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

Thread priority level

Set the priority for the thread. Higher-priority tasks can preempt lower-priority tasks.

Select one of the following five priority classes:

- `THREAD_PRIORITY_LOWEST`
- `THREAD_PRIORITY_BELOW_NORMAL`
- `THREAD_PRIORITY_NORMAL`
- `THREAD_PRIORITY_ABOVE_NORMAL`
- `THREAD_PRIORITY_HIGHEST`

Configuration Parameters for Simulink Models

- “Code Generation Pane: General” on page 4-2
- “Code Generation Pane: Report” on page 4-49
- “Code Generation Pane: Comments” on page 4-75
- “Code Generation Pane: Symbols” on page 4-102
- “Code Generation Pane: Custom Code” on page 4-150
- “Code Generation Pane: Debug” on page 4-167
- “Code Generation Pane: Interface” on page 4-177
- “Code Generation Pane: RSim Target” on page 4-266
- “Code Generation Pane: S-Function Target” on page 4-272
- “Code Generation Pane: Tornado Target” on page 4-278
- “Code Generation: Coder Target Pane” on page 4-309
- “Parameter Reference” on page 4-345

Code Generation Pane: General

The **Code Generation** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

To open the **Code Generation** pane, in the Simulink Editor, select **Simulation > Model Configuration Parameters > Code Generation**.

The screenshot displays the 'Code Generation' pane with the following sections and settings:

- Target selection:**
 - System target file: grt.tlc (with a 'Browse...' button)
 - Language: C (dropdown menu)
 - Description: Generic Real-Time Target
- Build process:**
 - Toolchain settings:**
 - Toolchain: Automatically locate an installed toolchain (dropdown menu) (with a 'Validate' button)
 - Build configuration: Faster Builds (dropdown menu) (with a 'Show settings' link)
 - Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)
 - Minimize compilation and linking time
- Code Generation Advisor:**
 - Select objective: Unspecified (dropdown menu)
 - Check model before generating code: Off (dropdown menu) (with a 'Check Model...' button)
- Actions:**
 - Generate code only (with a 'Build' button)
 - Package code and artifacts (with a 'Zip file name:' text box)

The **Code Generation** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

Target selection

System target file:

Language:

Description: Embedded Coder

Target hardware:

Build process

Toolchain settings

Toolchain:
Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)

Build configuration: [Show settings](#)
Minimize compilation and linking time

Data specification override

Ignore custom storage classes Ignore test point signals

Code Generation Advisor

Prioritized objectives: Unspecified

Check model before generating code:

Generate code only

Package code and artifacts Zip file name:

In this section...

“Code Generation: General Tab Overview” on page 4-5

“System target file” on page 4-6

“Browse” on page 4-8

“Language” on page 4-9

“Description” on page 4-11

“Target hardware” on page 4-12

“Toolchain” on page 4-14

“Build configuration” on page 4-16

“Tool/Options” on page 4-18

“Compiler optimization level” on page 4-19

“Custom compiler optimization flags” on page 4-21

“Generate makefile” on page 4-22

“Make command” on page 4-24

“Template makefile” on page 4-26

“Ignore custom storage classes” on page 4-28

“Ignore test point signals” on page 4-30

“Select objective” on page 4-32

“Prioritized objectives” on page 4-34

“Set Objectives” on page 4-35

“Set Objectives — Code Generation Advisor Dialog Box” on page 4-36

“Check Model” on page 4-39

“Check model before generating code” on page 4-40

“Generate code only” on page 4-42

“Build/Generate Code” on page 4-44

“Package code and artifacts” on page 4-45

“Zip file name” on page 4-47

Code Generation: General Tab Overview

Set up general information about code generation for a model's active configuration set, including target selection, documentation, and build process parameters.

To open the **Code Generation** pane, in the Simulink Editor, select **Simulation > Model Configuration Parameters > Code Generation**.

See Also

"Code Generation Pane: General" on page 4-2

System target file

Specify the system target file.

Settings

Default: `grt.tlc`

You can specify the system target file in these ways:

- Use the System Target File Browser. Click the **Browse** button, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command.
- Enter the name of your system target file in this field.

Setting **System target file** to `ert.tlc` displays the **Target hardware** parameter. When you set the **Target hardware** parameter to a specific type of hardware, the Configuration Parameters dialog box displays a **Coder Target** pane for that specific hardware. For more information, see “Target hardware” on page 4-12.

Tips

- The System Target File Browser lists system target files found on the MATLAB path. Some system target files require additional licensed products.
- Using ERT-based system target files such as `ert.tlc` to generate code requires an Embedded Coder license.
- To configure your model for rapid simulation, select `rsim.tlc`.
- To configure your model for Simulink Real-Time™, select `slrt.tlc` or `slrtert.tlc`.

Command-Line Information

Parameter: `SystemTargetFile`

Type: `string`

Value: valid system target file

Default: `'grt.tlc'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact (GRT) ERT based (requires Embedded Coder license)

See Also

“Available Targets”

Browse

Open the System Target File Browser, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command. The value you select is filled into “**System target file**” on page 4-6.

Tips

- The System Target File Browser lists system target files found on the MATLAB path. Some system target files require additional licensed products, such as the Embedded Coder product.
- To configure your model for rapid simulation, select `rsim.tlc`.
- To configure your model for Simulink Real-Time, select `slrt.tlc` or `slrtert.tlc`.

See Also

- “Select a Target”
- “Available Targets”

Language

Specify C or C++ code generation.

Settings

Default: C

C

Generates C code and places the generated files in your build folder.

C++

Generates C++ code and places the generated files in your build folder.

On the **Code Generation > Interface** pane, if you additionally set the **Code interface packaging** parameter to `C++ class`, the build generates a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.

If you set **Code interface packaging** to a value other than `C++ class`, the build generates C++ compatible `.cpp` files containing model interfaces enclosed within an `extern "C"` link directive.

You might need to configure the Simulink Coder software to use a compiler before you build a system.

Dependencies

Selecting C++ enables and selects the value `C++ class` for the **Code interface packaging** parameter on the **Code Generation > Interface** pane.

Command-Line Information

Parameter: TargetLang

Type: string

Value: 'C' | 'C++'

Default: 'C'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Change Programming Language”

“Compiler or IDE Selection and Configuration”

“Function Prototype Control”

“C++ Class Interface Control”

Description

This field displays the description of the system target file. You can use this description to differentiate between two system target files that have the same file name. To change the value of this description, click the Browse button.

See Also

“Browse” on page 4-8

Target hardware

Select the target hardware for which to generate code.

Note

- This parameter only appears when the model is configured to use the toolchain approach, as described in “Configure the Build Process”
 - Using this parameter to generate code requires an Embedded Coder license.
-

To use the **Target hardware** parameter, both of the following actions must be complete:

- Set the **System target file** parameter to `ert.tlc`. This action makes the **Target hardware** parameter visible on the **Code Generation** pane.
- Use Support Package Installer to install the Embedded Coder support package for your target hardware. This action makes target hardware options available for the **Target hardware** parameter.

To install support for your target hardware, set **Target hardware** to **Get more**. This action opens Support Package Installer and displays a list of the support packages that are available for Embedded Coder software. Install one of the following support packages:

- “Support Package for ARM® Cortex®-M Processors”
- “Support Package for STMicroelectronics STM32F4 Discovery™ Board”
- “Support Package for Texas Instruments C2000 Processors ”

When you set the **Target hardware** parameter to one of the target hardware options, the Configuration Parameters dialog box displays one of the following **Coder Target** panes:

- “Coder Target Pane: ARM Cortex-M3 (QEMU)”
- “Coder Target Pane: Support Package for STMicroelectronics® STM32F4 Discovery Hardware”

- “Coder Target Pane: Texas Instruments C2000™ Processors”

Settings

Default: None

None

Target hardware not specified.

Get more...

Select **Get more...** to start Support Package Installer and install Embedded Coder support packages. Embedded Coder support packages add options to the **Target hardware** parameter.

Command-Line Information

Parameter: Not available

Type: Not available

Value: Not available

Default: Not available

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Toolchain”
- “Adding a Custom Toolchain”

Toolchain

Specify the toolchain to use when building an executable or library.

Note This parameter only appears when the model is configured to use the toolchain approach, as described in “Configure the Build Process”

Settings

Default: Automatically locate an installed toolchain

The list of available toolchains depends on the host computer platform, and can include custom toolchains that you added.

When **Toolchain** is set to Automatically locate an installed toolchain, the coder software:

- 1 Searches your host computer for installed toolchains.
- 2 Selects the most current toolchain.
- 3 Displays the name of the selected toolchain immediately below the drop down menu.

Tip

Click **Validate** to verify that the registration information for the toolchain is valid. When the validation process is complete, a separate **Validation report** window opens and displays the results. The Validation report states whether the toolchain registration Passed or Failed and provides status for each step in the validation process. To fix a failure, edit the toolchain definition and repeat the registration process.

Command-Line Information

Parameter: Toolchain

Type: string

Value: 'Automatically locate an installed toolchain' | A valid toolchain name

Default: 'Automatically locate an installed toolchain'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Toolchain”
- “Adding a Custom Toolchain”

Build configuration

Specify compiler optimization or debug settings for toolchain.

Note This parameter only appears when the model is configured to use the toolchain approach, as described in “Configure the Build Process”

Settings

Default: Faster Builds

Faster Builds

Optimize for shorter build times.

Faster Runs

Optimize for faster-running executable.

Debug

Optimize for debugging.

Specify

Selecting **Specify** displays a table of tools with editable options. Use this table to customize settings for the current model. See “Tool/Options” on page 4-18.

Tip

Click **Show settings** to display a table of tools with options for the current build configuration. See “Tool/Options” on page 4-18.

Customize the toolchain options for the **Specify** build configuration. These options only apply to the current project.

Dependencies

Selecting **Specify** displays a table of tools with editable options. Use this table to customize settings for the current model. See “Tool/Options” on page 4-18.

Command-Line Information

Parameter: BuildConfiguration

Type: string

Value: 'Faster Builds' | 'Faster Runs' | 'Debug' | 'Specify'

Default: 'Faster Builds'

Recommended Settings

Application	Setting
Debugging	Debug
Traceability	No impact
Efficiency	Faster Runs
Safety precaution	No impact

See Also

- “Toolchain”
- “Adding a Custom Toolchain”

Tool/Options

Display or customize build configuration settings.

Note These parameters only appear when the model is configured to use the toolchain approach, as described in “Configure the Build Process”

Settings

The tools column can include: Assembler, C Compiler, Linker, Shared Library Linker, C++ Compiler, C++ Linker, C++ Shared Library Linker, Archiver, Download, Execute, Make Tool. The options can vary by tool and toolchain and can contain macros. Consult third-party toolchain documentation for more information about options you can use with a specific tool.

Dependencies

To display a table of tools and options for the current build configuration, click **Show settings**, next to **Build configuration**.

To create a custom build configuration by editing a table of Tool/Options, set **Build configuration** to Specify.

Command-Line Information

Parameter: CustomToolchainOptions

Type: string

Value: Specify the baseline toolchain settings. Use a new-line-delineated string to specify each option and its values.

Default: ''

See Also

- “Toolchain”
- “Adding a Custom Toolchain”

Compiler optimization level

Control compiler optimizations for building generated code, using flexible, generalized controls.

Note This parameter only appears when the model is configured to use the template makefile approach, as described in “Configure the Build Process”

Settings

Default: Optimizations off (faster builds)

Optimizations off (faster builds)

Customizes compilation during the Simulink Coder makefile build process to minimize compilation time.

Optimizations on (faster runs)

Customizes compilation during the Simulink Coder makefile build process to minimize run time.

Custom

Allows you to specify custom compiler optimization flags to be applied during the Simulink Coder makefile build process.

Tips

- Target-independent values **Optimizations on (faster runs)** and **Optimizations off (faster builds)** allow you to easily toggle compiler optimizations on and off during code development.
- **Custom** allows you to enter custom compiler optimization flags at Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to Simulink Coder make commands.
- If you specify compiler options for your Simulink Coder makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS=" -v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

Dependencies

This parameter enables **Custom compiler optimization flags**.

Command-Line Information

Parameter: RTWCompilerOptimization

Type: string

Value: 'Off' | 'On' | 'Custom'

Default: 'Off'

Recommended Settings

Application	Setting
Debugging	Optimizations off (faster builds)
Traceability	Optimizations off (faster builds)
Efficiency	Optimizations on (faster runs) (execution), No impact (ROM, RAM)
Safety precaution	No impact

See Also

- “Custom compiler optimization flags” on page 4-21
- “Control Compiler Optimizations”

Custom compiler optimization flags

Specify compiler optimization flags to be applied to building the generated code for your model.

Note This parameter only appears when the model is configured to use the template makefile approach, as described in “Configure the Build Process”

Settings

Default: ''

Specify compiler optimization flags without quotes, for example, -O2.

Dependency

This parameter is enabled by selecting the value `Custom` for the parameter **Compiler optimization level**.

Command-Line Information

Parameter: RTWCustomCompilerOptimizations

Type: string

Value: '' | user-specified flags

Default: ''

Recommended Settings

See “Compiler optimization level” on page 4-19.

See Also

- “Compiler optimization level” on page 4-19
- “Control Compiler Optimizations”

Generate makefile

Enable generation of a makefile based on a template makefile.

Note This parameter only appears when the model is configured to use the template makefile approach, as described in “Configure the Build Process”

Settings

Default: on



On

Generates a makefile for a model during the build process.



Off

Suppresses the generation of a makefile. You must set up post code generation build processing, including compilation and linking, as a user-defined command.

Dependencies

This parameter enables:

- **Make command**
- **Template makefile**

Command-Line Information

Parameter: GenerateMakefile

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Customize Post-Code-Generation Build Processing”
- “Customize Build Process with STF_make_rtw_hook File”
- “Target Development and the Build Process”

Make command

Specify a make command and optionally append make command arguments.

Note This parameter only appears when the model is configured to use the template makefile approach, as described in “Configure the Build Process”

Settings

Default: `make_rtw`

The make command, a high-level MATLAB command, invoked when you start a build, controls the Simulink Coder build process.

- Each target has an associated make command, automatically supplied when you select a target file using the System Target File Browser.
- Some third-party targets supply a make command. See the vendor’s documentation.
- You can specify arguments in the **Make command** field which pass into the makefile-based build process. Append the arguments after the make command, as in the following example:

```
make_rtw OPTS=" -DMYDEFINE=1 "
```

The syntax for make command options differs slightly for different compilers.

Tip

Most targets use the default command.

Dependency

This parameter is enabled by **Generate makefile**.

Command-Line Information

Parameter: MakeCommand

Type: string

Value: valid make command MATLAB language file

Default: 'make_rtw'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	make_rtw

See Also

- “Template Makefiles and Make Options”
- “Customize Build Process with STF_make_rtw_hook File”
- “Target Development and the Build Process”

Template makefile

Specify the template makefile from which to generate the makefile.

Note This parameter only appears when the model is configured to use the template makefile approach, as described in “Configure the Build Process”

Settings

Default: grt_default_tmf

The template makefile determines which compiler runs, during the make phase of the build, to compile the generated code. You can specify template makefiles in the following ways:

- Generate a value by selecting a target configuration using the System Target File Browser.
- Explicitly enter a custom template makefile filename (including the extension). The file must be on the MATLAB path.

Tips

- If you do not include a filename extension for a custom template makefile, the code generator attempts to find and execute a MATLAB language file.
- You can customize your build process by modifying an existing template makefile or by providing your own template makefile.

Dependency

This parameter is enabled by **Generate makefile**.

Command-Line Information

Parameter: TemplateMakefile

Type: string

Value: valid template makefile filename

Default: 'grt_default_tmf'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Template Makefiles and Make Options”
- “Available Targets”

Ignore custom storage classes

Specify whether to apply or ignore custom storage classes.

Settings

Default: off



On

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



Off

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

Tips

- Clear this parameter before configuring data objects with custom storage classes.
- Setting for top-level and referenced models must match.

Dependencies

- This parameter only appears for ERT-based targets.
- Clear this parameter to enable module packaging features.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: IgnoreCustomStorageClasses

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Custom Storage Classes” in the Embedded Coder documentation

Ignore test point signals

Specify allocation of memory buffers for test points.

Settings

Default: Off



On

Ignores test points during code generation, allowing optimal buffer allocation for signals with test points, facilitating transition from prototyping to deployment and avoiding accidental degradation of generated code due to workflow artifacts.



Off

Allocates separate memory buffers for test points, resulting in a loss of code generation optimizations such as reducing memory usage by storing signals in reusable buffers.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: IgnoreTestpoints

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact

Application	Setting
Efficiency	On
Safety precaution	No impact

See Also

- “Signals with Test Points” in the Simulink Coder documentation
- “Test Points” in the Simulink documentation
- “Signals” in the Simulink Coder documentation

Select objective

Select code generation objectives to use with the Code Generation Advisor.

Settings

Default: Unspecified

Unspecified

No objective specified. Do not optimize code generation settings using the Code Generation Advisor.

Debugging

Specifies debugging objective. Optimize code generation settings for debugging the code generation build process using the Code Generation Advisor.

Execution efficiency

Specifies execution efficiency objective. Optimize code generation settings to achieve fast execution time using the Code Generation Advisor.

Tips

For more objectives, specify an ERT-based target.

Dependency

These parameters appear only for GRT-based targets.

Command-Line Information

Parameter: 'ObjectivePriorities'

Type: cell array of strings

Value: {' '} | {'Debugging'} | {'Execution efficiency'}

Default: {' '}

Recommended Settings

Application	Setting
Debugging	Debugging
Traceability	Not applicable for GRT-based targets
Efficiency	Execution efficiency
Safety precaution	Not applicable for GRT-based targets

See Also

- “Application Objectives” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder documentation.

Prioritized objectives

List objectives that you specify by clicking the **Set Objectives** button.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Command: `get_param('model', 'ObjectivePriorities')`

See Also

- “Application Objectives” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder documentation.

Set Objectives

Open Configuration Set Objectives dialog box.

Dependency

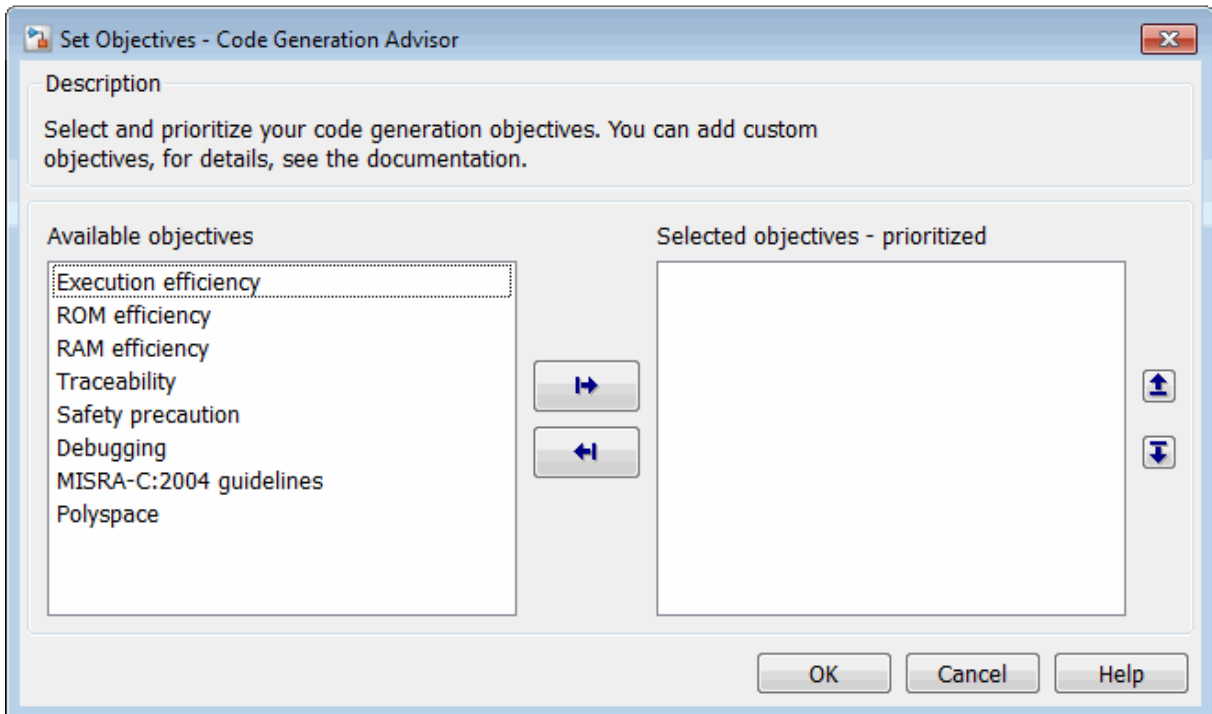
This button appears only for ERT-based targets.

See Also

- “Application Objectives” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder documentation.

Set Objectives – Code Generation Advisor Dialog Box

Select and prioritize code generation objectives to use with the Code Generation Advisor.



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:2004 guidelines — Configure code generation settings to increase compliance with MISRA-C:2004 guidelines.

Polyspace — Configure code generation settings to prepare the code for Polyspace® analysis.

Note If you select the MISRA-C:2004 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA-C:2004 configuration setting recommendations.
- For blocks that are not supported or recommended for MISRA-C:2004 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 strings; combination of the available values

Value: {' '} | {'Execution efficiency'} | {'ROM efficiency'} | {'RAM efficiency'} | {'Traceability'} | {'Safety precaution'} | {'Debugging'} | {'MISRA-C:2004 guidelines'} | {'Polyspace'}

Default: {' '}

See Also

- “Application Objectives” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder documentation.

Check Model

Run the Code Generation Advisor checks.

Settings

- 1 Specify code generation objectives using the **Select objective** parameter (available with GRT-based targets) or in the Configuration Set Objectives dialog box, by clicking **Set Objectives** (available with ERT-based targets).
- 2 Click **Check Model**. The Code Generation Advisor runs the code generation objectives checks and provide suggestions for changing your model to meet the objectives.

Dependency

You must specify objectives before checking the model.

See Also

- “Application Objectives” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder documentation.

Check model before generating code

Choose whether to run Code Generation Advisor checks before generating code.

Settings

Default: Off

Off

Generates code without checking whether the model meets code generation objectives. The code generation report summary and file headers indicate the specified objectives and that the validation was not run.

On (proceed with warnings)

Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the Simulink Coder software continues generating code. The code generation report summary and file headers indicate the specified objectives and the validation result.

On (stop for warnings)

Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the Simulink Coder software does not continue generating code.

Command-Line Information

Parameter: CheckMdlBeforeBuild

Type: string

Value: 'Off' | 'Warning' | 'Error'

Default: 'Off'

Recommended Settings

Application	Setting
Debugging	On (proceed with warnings) or On (stop for warnings)
Traceability	On (proceed with warnings) or On (stop for warnings)
Efficiency	On (proceed with warnings) or On (stop for warnings)
Safety precaution	On (proceed with warnings) or On (stop for warnings)

See Also

- “Application Objectives” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder documentation.

Generate code only

Specify code generation versus an executable build.

Settings

Default: off



On

The caption of the **Build/Generate Code** button becomes **Generate Code**. The build process generates code and a makefile, but it does not invoke the make command.



Off

The caption of the **Build/Generate Code** button becomes **Build**. The build process generates and compiles code, and creates an executable file.

Tip

Generate code only generates a makefile only if you select **Generate makefile**.

Dependency

This parameter changes the function of the **Build/Generate Code** button.

Command-Line Information

Parameter: GenCodeOnly

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

“Customize Post-Code-Generation Build Processing”

Build/Generate Code

Start the build or code generation process.

Tip

You can also start the build process by pressing **Ctrl+B**.

Dependency

When you select **Generate code only**, the caption of the **Build** button changes to **Generate Code**.

Command-Line Information

Command: `rtwbuild`

Type: `string`

Value: `'modelName'`

Recommended Settings

Application	Setting
Debugging	Build
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Initiate the Build Process”

Package code and artifacts

Specify whether to automatically package generated code and artifacts for relocation.

Settings

Default: off



On

The build process runs the packNGo function after code generation to package generated code and artifacts for relocation.



Off

The build process does not run packNGo after code generation.

Dependency

Selecting this parameter enables **Zip file name** and clearing this parameter disables **Zip file name**.

Command-Line Information

Parameter: PackageGeneratedCodeAndArtifacts

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Relocate Code to Another Development Environment”
- “packNGo Function Limitations”

Zip file name

Specify the name of the .zip file in which to package generated code and artifacts for relocation.

Settings

Default: ''

You can enter the name of the zip file in which to package generated code and artifacts for relocation. The file name can be specified with or without the .zip extension. If you do not specify an extension or an extension other than .zip, the zip utility adds the .zip extension. If a value is not specified, the build process uses the name *model.zip*, where *model* is the name of the top model for which code is being generated.

Dependency

This parameter is enabled by **Package code and artifacts**.

Command-Line Information

Parameter: PackageName

Type: string

Value: valid name for a .zip file

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Relocate Code to Another Development Environment”
- “packNGo Function Limitations”

Code Generation Pane: Report

The **Code Generation > Report** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

- Create code generation report Open report automatically

The **Code Generation > Report** pane includes the following additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

Navigation

Code-to-model

Model-to-code

Generate model Web view

Traceability Report Contents

Eliminated / virtual blocks

Traceable Simulink blocks

Traceable Stateflow objects

Traceable MATLAB functions

Metrics

Static code metrics

Summarize which blocks triggered code replacements

In this section...

“Code Generation: Report Tab Overview” on page 4-51

“Create code generation report” on page 4-52

“Open report automatically” on page 4-55

In this section...

“Code-to-model” on page 4-57

“Model-to-code” on page 4-59

“Configure” on page 4-61

“Generate model Web view” on page 4-62

“Eliminated / virtual blocks” on page 4-63

“Traceable Simulink blocks” on page 4-65

“Traceable Stateflow objects” on page 4-67

“Traceable MATLAB functions” on page 4-69

“Static code metrics” on page 4-71

“Summarize which blocks triggered code replacements” on page 4-73

Code Generation: Report Tab Overview

Control the code generation report that the Simulink Coder software automatically creates.

Configuration

To create a code generation report during the build process, select the **Create code generation report** parameter.

See Also

- “Generate a Code Generation Report”
- “Reports for Code Generation”

If you have an Embedded Coder license, see also “HTML Code Generation Report Extensions”.

- “Code Generation Pane: Report” on page 4-49

Create code generation report

Document generated code in an HTML report.

Settings

Default: Off



On

Generates a summary of code generation source files in an HTML report. Places the report files in an `html` subfolder within the build folder. In the report,

- The **Summary** section lists version and date information. The **Configuration Settings at the Time of Code Generation** link opens a noneditable view of the Configuration Parameters dialog that shows the Simulink model settings, including TLC options, at the time of code generation.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data (requires an Embedded Coder license and the ERT target).
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**, providing a complete mapping between model elements and code (requires an Embedded Coder license and the ERT target).
- The **Static Code Metrics Report** section provides statistics of the generated code. Metrics are estimated from static analysis of the generated code.
- The **Code Replacements Report** section allows you to account for code replacement library (CRL) functions that were used during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement.

In the **Generated Files** section, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code,

- Global variable instances are hyperlinked to their definitions.
- If you selected the traceability option **Code-to-model**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view the relevant blocks or subsystems in a Simulink model window (requires an Embedded Coder license and the ERT target).
- If you selected the traceability option **Model-to-code**, you can view the generated code for a block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **C/C++ Code > Navigate to C/C++ Code** (requires an Embedded Coder license and the ERT target).
- If you set the **Code coverage tool** parameter on the **Code Generation > Verification** pane, you can view the code coverage data and annotations in the generated code in the HTML Code Generation Report (requires an Embedded Coder license and the ERT target).



Off

Does not generate a summary of files.

Dependency

This parameter enables and selects

- “Open report automatically” on page 4-55
- “Code-to-model” on page 4-57 (ERT target)

This parameter enables

- “Model-to-code” on page 4-59 (ERT target)
- “Eliminated / virtual blocks” on page 4-63 (ERT target)
- “Traceable Simulink blocks” on page 4-65 (ERT target)
- “Traceable Stateflow objects” on page 4-67 (ERT target)

- “Traceable MATLAB functions” on page 4-69 (ERT target)

Command-Line Information

Parameter: GenerateReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“Reports for Code Generation”

If you have an Embedded Coder license, see also “HTML Code Generation Report Extensions”.

If you have an Embedded Coder license, see also “Configure SIL and PIL Code Coverage”.

Open report automatically

Specify whether to display code generation reports automatically.

Settings

Default: Off



On

Displays the code generation report automatically in a new browser window.



Off

Does not display the code generation report, but the report is still available in the html folder.

Dependency

This parameter is enabled and selected by **Create code generation report**.

Command-Line Information

Parameter: LaunchReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

“Reports for Code Generation”

If you have an Embedded Coder license, see also “HTML Code Generation Report Extensions”.

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.

Settings

Default: Off



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.

Tip

Clear this parameter to speed up code generation. For large models (containing over 1000 blocks), generation of hyperlinks can be time consuming.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled and selected by **Create code generation report**.
- You must select **Include comments** on the **Code Generation > Comments** pane to use this parameter.

Command-Line Information

Parameter: IncludeHyperlinkInReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“HTML Code Generation Report Extensions”

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.

Settings

Default: Off



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 and select **C/C++ Code > Navigate to C/C++ Code**.



Off

Omits model-to-code highlighting support from the generated report.

Tip

Clear this parameter to speed up code generation. For large models (containing over 1000 blocks), generation of model-to-code highlighting support can be time consuming.

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**.
- This parameter selects:
 - “Eliminated / virtual blocks” on page 4-63
 - “Traceable Simulink blocks” on page 4-65
 - “Traceable Stateflow objects” on page 4-67
 - “Traceable MATLAB functions” on page 4-69
- You must select the following parameters to use this parameter:

- “Include comments” on page 4-79 on the **Code Generation > Comments** pane
- At least one of the following:
 - “Eliminated / virtual blocks” on page 4-63
 - “Traceable Simulink blocks” on page 4-65
 - “Traceable Stateflow objects” on page 4-67
 - “Traceable MATLAB functions” on page 4-69

Command-Line Information

Parameter: GenerateTraceInfo

Type: Boolean

Value: on | off

Default: off

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“HTML Code Generation Report Extensions”

Configure

Open the **Model-to-code navigation** dialog box. This dialog box provides a way for you to specify a build folder containing previously-generated model code to highlight. Applying your build folder selection will attempt to load traceability information from the earlier build, for which **Model-to-code** must have been selected.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by “Model-to-code” on page 4-59.

See Also

“HTML Code Generation Report Extensions”

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. 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 of the model in the code generation report.

Settings

Default: Off



On

Include model Web view in the code generation report.



Off

Omit model Web view in the code generation report.

Command-Line Information

Parameter: GenerateWebview

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Web View of Model in Code Generation Report”

Eliminated / virtual blocks

Include summary of eliminated and virtual blocks in code generation report.

Settings

Default: Off

- On
Includes a summary of eliminated and virtual blocks in the code generation report.
- Off
Does not include a summary of eliminated and virtual blocks.

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**.
- This parameter is selected by **Model-to-code**.

Command-Line Information

Parameter: GenerateTraceReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“HTML Code Generation Report Extensions”

Traceable Simulink blocks

Include summary of Simulink blocks in code generation report.

Settings

Default: Off



On

Includes a summary of Simulink blocks and the corresponding code location in the code generation report.



Off

Does not include a summary of Simulink blocks.

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**.
- This parameter is selected by **Model-to-code**.

Command-Line Information

Parameter: GenerateTraceReportSl

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“HTML Code Generation Report Extensions”

Traceable Stateflow objects

Include summary of Stateflow objects in code generation report.

Settings

Default: Off



On

Includes a summary of Stateflow objects and the corresponding code location in the code generation report.



Off

Does not include a summary of Stateflow objects.

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**.
- This parameter is selected by **Model-to-code**.

Command-Line Information

Parameter: GenerateTraceReportSf

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“HTML Code Generation Report Extensions”

“Traceability of Stateflow Objects in Generated Code”

Traceable MATLAB functions

Include summary of MATLAB functions in code generation report.

Settings

Default: Off



On

Includes a summary of MATLAB functions and corresponding code locations in the code generation report.



Off

Does not include a summary of MATLAB functions.

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**.
- This parameter is selected by **Model-to-code**.

Command-Line Information

Parameter: GenerateTraceReportEm1

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“HTML Code Generation Report Extensions”

Static code metrics

Include static code metrics report in the code generation report.

Settings

Default: Off

- On
Include static code metrics report in the code generation report. The static code metrics report does not support the C++ target language.
- Off
Omit static code metrics report from 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**.

Command-Line Information

Parameter: GenerateCodeMetricsReport

Type: Boolean

Value: on | off

Default: off

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Static Code Metrics”

Summarize which blocks triggered code replacements

Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

Settings

Default: Off



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.

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**.

Command-Line Information

Parameter: GenerateCodeReplacementReport

Type: Boolean

Value: on | off

Default: off

Recommended Settings

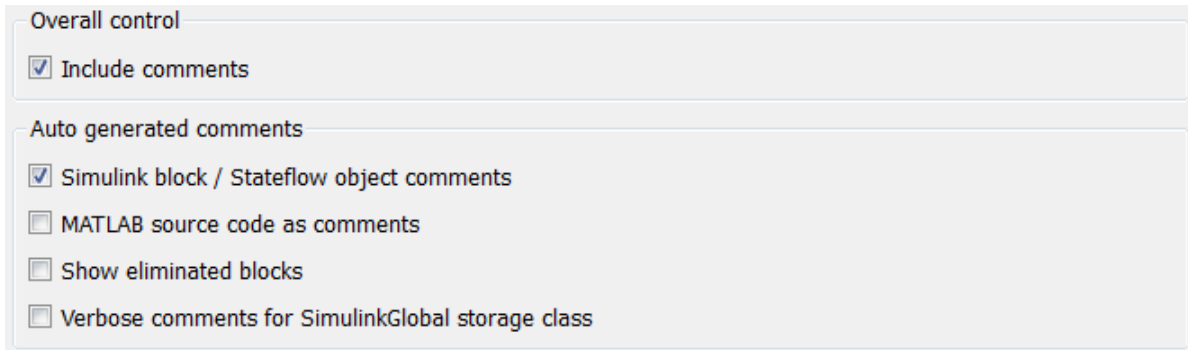
Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- Analyze Code Replacements in the Generated Code
- Trace Code Replacements Generated Using Your Code Replacement Library
- Determine Why Code Replacement Functions Were Not Used

Code Generation Pane: Comments

The **Code Generation > Comments** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.



The screenshot shows a settings pane with two sections. The first section, 'Overall control', contains a checked checkbox for 'Include comments'. The second section, 'Auto generated comments', contains four unchecked checkboxes: 'Simulink block / Stateflow object comments', 'MATLAB source code as comments', 'Show eliminated blocks', and 'Verbose comments for SimulinkGlobal storage class'.

Section	Parameter	Checked
Overall control	Include comments	Yes
Auto generated comments	Simulink block / Stateflow object comments	No
	MATLAB source code as comments	No
	Show eliminated blocks	No
	Verbose comments for SimulinkGlobal storage class	No

The **Code Generation > Comments** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

Overall control

Include comments

Auto generated comments

Simulink block / Stateflow object comments

MATLAB source code as comments

Show eliminated blocks

Verbose comments for SimulinkGlobal storage class

Operator annotations

Custom comments

Simulink block descriptions Stateflow object descriptions

Simulink data object descriptions Requirements in block comments

Custom comments (MPT objects only) MATLAB function help text

In this section...

“Code Generation: Comments Tab Overview” on page 4-78

“Include comments” on page 4-79

“Simulink block / Stateflow object comments” on page 4-81

“MATLAB source code as comments” on page 4-82

“Show eliminated blocks” on page 4-84

“Verbose comments for SimulinkGlobal storage class” on page 4-85

“Operator annotations” on page 4-86

“Simulink block descriptions” on page 4-88

“Simulink data object descriptions” on page 4-90

“Custom comments (MPT objects only)” on page 4-92

“Custom comments function” on page 4-94

“Stateflow object descriptions” on page 4-96

In this section...

“Requirements in block comments” on page 4-98

“MATLAB function help text” on page 4-100

Code Generation: Comments Tab Overview

Control the comments that the Simulink Coder software automatically creates and inserts into the generated code.

See Also

“Code Generation Pane: Comments” on page 4-75

Include comments

Specify which comments are in generated files.

Settings

Default: on



On

Places comments in the generated files based on the selections in the **Auto generated comments** pane.



Off

Omits comments from the generated files.

Note This parameter does not apply to copyright notice comments in the generated code.

Dependencies

This parameter enables:

- **Simulink block / Stateflow object comments**
- **MATLAB source code as comments**
- **Show eliminated blocks**
- **Verbose comments for SimulinkGlobal storage class**

Command-Line Information

Parameter: GenerateComments

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

Simulink block / Stateflow object comments

Specify whether to insert Simulink block and Stateflow object comments.

Settings

Default: on



On

Inserts automatically generated comments that describe a block's code and objects. The comments precede that code in the generated file.



Off

Suppresses comments.

Dependency

This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: SimulinkBlockComments

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

MATLAB source code as comments

Specify whether to insert MATLAB source code as comments.

Settings

Default: off



On

Inserts MATLAB source code as comments in the generated code. The comments precede the associated generated code.

Includes the function signature in the function banner.



Off

Suppresses comments and does not include the function signature in the function banner.

Dependency

This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: MATLABSourceComments

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“Include MATLAB Code as Comments in Generated Code”

Show eliminated blocks

Specify whether to insert eliminated block's comments.

Settings

Default: off



On

Inserts statements in the generated code from blocks eliminated as the result of optimizations (such as parameter inlining).



Off

Suppresses statements.

Dependency

This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: ShowEliminatedStatement

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

Verbose comments for SimulinkGlobal storage class

You can control the generation of comments in the model parameter structure declaration in *model_prm.h*. Parameter comments indicate parameter variable names and the names of source blocks.

Settings

Default: off



On

Generates parameter comments regardless of the number of parameters.



Off

Generates parameter comments if less than 1000 parameters are declared. This reduces the size of the generated file for models with a large number of parameters.

Dependency

This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: ForceParamTrailComments

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

Operator annotations

Specify whether to include operator annotations for Polyspace in the generated code as comments.

Settings

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 Embedded Coder software 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 Embedded Coder software uses one of these operators, it adds annotations to the generated code to justify the operators for Polyspace.
- Embedded Coder cannot justify operators that result from the design.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: OperatorAnnotations

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“Code Annotation for Justifying Polyspace Checks”

Simulink block descriptions

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

Settings

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 / Stateflow object comments**
- Text specified in the **Description** field of each Block Properties dialog box

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



Off

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

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: InsertBlockDesc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

“International Character Support”

Simulink data object descriptions

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

Settings

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.

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



Off

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

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: SimulinkDataObjDesc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

Custom comments (MPT objects only)

Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.

Settings

Default: off



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.

Dependency

- 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**.

Command-Line Information

Parameter: EnableCustomComments

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

“Add Custom Comments for Signal or Parameter Identifiers”

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

Settings

Default: ''

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.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Custom comments (MPT objects only)**.

Command-Line Information

Parameter: CustomCommentsFcn

Type: string

Value: valid file name

Default: ''

Recommended Settings

Application	Setting
Debugging	Valid file name
Traceability	Valid file name

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

“Add Custom Comments for Signal or Parameter Identifiers”

Stateflow object descriptions

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

Settings

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.

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



Off

Suppresses the generation of comments for Stateflow objects.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Stateflow license.

Command-Line Information

Parameter: SFDataObjDesc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

“International Character Support”

Requirements in block comments

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

Settings

Default: off



On

Inserts the requirement descriptions that you assign to Simulink blocks into the generated code as comments. The Simulink Coder software 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

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



Off

Suppresses the generation of comments for block requirement descriptions.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires Embedded Coder and Simulink Verification and Validation™ licenses when generating code.

Command-Line Information

Parameter: ReqsInCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

“How Requirements Information Is Included in Generated Code” in the Simulink Verification and Validation documentation

MATLAB function help text

Specify whether to include MATLAB function help text in the function banner.

Settings

Default: off

- On
Inserts MATLAB function help text in the function banner.
- Off
Inserts MATLAB function help text in the body of the function.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: MATLABFcnDesc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

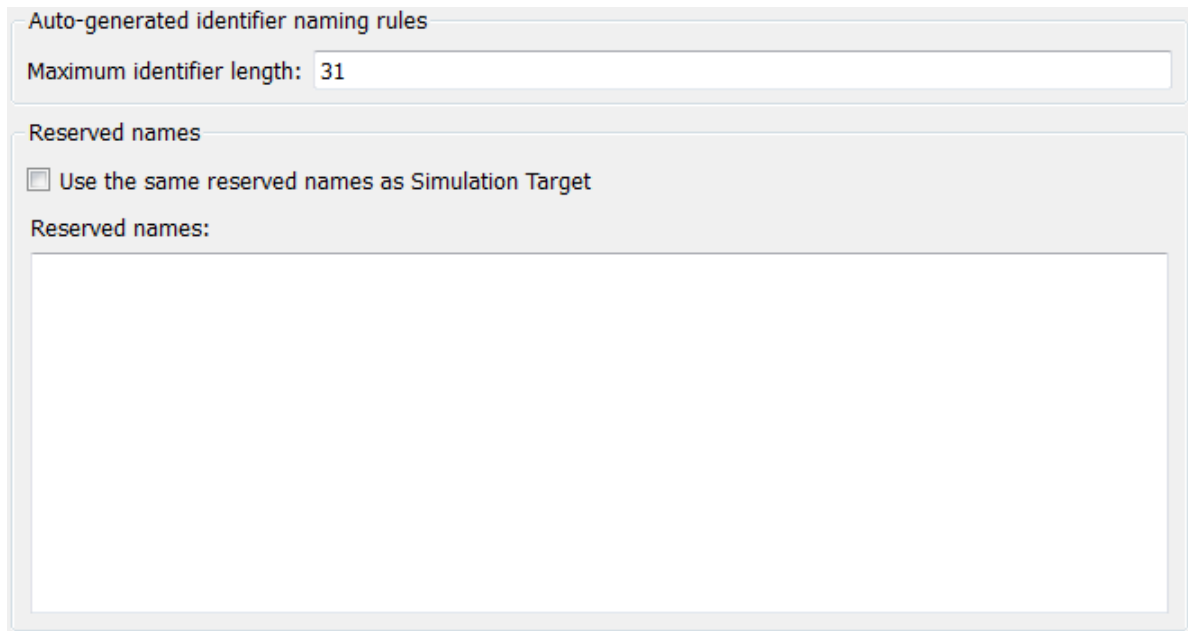
Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

“Including MATLAB Function Help Text in the Function Banner”

Code Generation Pane: Symbols

The **Code Generation > Symbols** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.



The screenshot shows a configuration pane with two sections. The first section, titled "Auto-generated identifier naming rules", contains a text input field labeled "Maximum identifier length:" with the value "31". The second section, titled "Reserved names", contains a checkbox labeled "Use the same reserved names as Simulation Target" which is currently unchecked. Below the checkbox is a label "Reserved names:" followed by a large, empty rectangular text area.

The **Code Generation > Symbols** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

Auto-generated identifier naming rules

Identifier format control

Global variables:	<input type="text" value="\$R\$N\$M"/>
Global types:	<input type="text" value="\$N\$R\$M_T"/>
Field name of global types:	<input type="text" value="\$N\$M"/>
Subsystem methods:	<input type="text" value="\$R\$N\$M\$F"/>
Subsystem method arguments:	<input type="text" value="rt\$I\$N\$M"/>
Local temporary variables:	<input type="text" value="\$N\$M"/>
Local block output variables:	<input type="text" value="rtb_-\$N\$M"/>
Constant macros:	<input type="text" value="\$R\$N\$M"/>
Shared utilities:	<input type="text" value="\$N\$C"/>

Minimum mangle length:

Maximum identifier length:

System-generated identifiers:

Generate scalar inlined parameters as:

Simulink data object naming rules

Signal naming:

Parameter naming:

#define naming:

Reserved names

Use the same reserved names as Simulation Target

Reserved names:

In this section...

- “Code Generation: Symbols Tab Overview” on page 4-105
- “Global variables” on page 4-106
- “Global types” on page 4-109
- “Field name of global types” on page 4-112
- “Subsystem methods” on page 4-114
- “Subsystem method arguments” on page 4-117
- “Local temporary variables” on page 4-119
- “Local block output variables” on page 4-122
- “Constant macros” on page 4-124
- “Shared utilities” on page 4-127
- “Minimum mangle length” on page 4-128
- “Maximum identifier length” on page 4-131
- “System-generated identifiers” on page 4-133
- “Generate scalar inlined parameter as” on page 4-138
- “Signal naming” on page 4-139
- “M-function” on page 4-141
- “Parameter naming” on page 4-143
- “#define naming” on page 4-145
- “Use the same reserved names as Simulation Target” on page 4-147
- “Reserved names” on page 4-148

Code Generation: Symbols Tab Overview

Select the automatically generated identifier naming rules.

See Also

- “Code Generation Pane: Symbols” on page 4-102
- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Specify Identifier Length to Avoid Naming Collisions”
- “Specify Reserved Names for Generated Identifiers”
- “Customize Generated Identifier Naming Rules” in the Embedded Coder documentation

Global variables

Customize generated global variable identifiers.

Settings

Default: \$R\$N\$M

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

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

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 a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model,

the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

- This parameter setting only determines the name of objects, such as signals and parameters, if the object is set to Auto.

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrGlobalVar

Type: string

Value: valid combination of tokens

Default: \$R\$N\$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	\$R\$N\$M

See Also

- “Identifier Format Control” in the Embedded Coder documentation
- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Avoid Identifier Name Collisions with Referenced Models” in the Embedded Coder documentation

- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Global types

Customize generated global type identifiers.

Settings

Default: \$N\$R\$M_T

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

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

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 a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model,

the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `CustomSymbolStrType`

Type: `string`

Value: valid combination of tokens

Default: `NR$M_T`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	<code>\$N\$R\$M_T</code>

See Also

- “Identifier Format Control” in the Embedded Coder documentation

- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Avoid Identifier Name Collisions with Referenced Models” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Field name of global types

Customize generated field names of global types.

Settings

Default: \$N\$M

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

Token	Description
\$A	Insert data type acronym 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 string <code>root_</code> . For blocks at the subsystem level, the tag is of the form <code>sN_</code> , where N is a unique system number assigned by the Simulink software.
\$M	Insert name mangling string if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

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 a name mangling string.
- 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).

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrField

Type: string

Value: valid combination of tokens

Default: \$N\$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	\$N\$M

See Also

- “Identifier Format Control” in the Embedded Coder documentation
- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Subsystem methods

Customize generated function names for reusable subsystems.

Settings

Default: \$R\$N\$M\$F

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

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

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrFcn

Type: string

Value: valid combination of tokens

Default: \$R\$N\$M\$F

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens

Application	Setting
Efficiency	No impact
Safety precaution	\$R\$N\$M\$F

See Also

- “Identifier Format Control” in the Embedded Coder documentation
- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Avoid Identifier Name Collisions with Referenced Models” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Subsystem method arguments

Customize generated function argument names for reusable subsystems.

Settings

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

Token	Description
\$I	Insert an u if the argument is an input. Insert a y if the argument is an output. Optional.
\$M	Insert name mangling string if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. Recommended to maximize readability of generated code.

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 a name mangling string.

Dependencies

This parameter:

- Appears only for ERT-based targets.

- Requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrFcnArg

Type: string

Value: valid combination of tokens

Default: rtu_\$\$M or rty_\$\$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combinations of tokens
Efficiency	No impact
Safety precaution	rtu_\$\$M or rty_\$\$M

See Also

- “Code Generation Pane: Symbols” on page 4-102
- “Identifier Format Control” in the Embedded Coder documentation
- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Local temporary variables

Customize generated local temporary variable identifiers.

Settings

Default: \$N\$M

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

Token	Description
\$A	Insert data type acronym (for example, i32 for integers) into signal and work vector identifiers.
\$M	Insert name mangling string if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated.
\$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

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 a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.

- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This option does not impact objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrTmpVar

Type: string

Value: valid combination of tokens

Default: \$N\$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	\$N\$M

See Also

- “Identifier Format Control” in the Embedded Coder documentation
- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation

- “Avoid Identifier Name Collisions with Referenced Models” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Local block output variables

Customize generated local block output variable identifiers.

Settings

Default: `rtb_$$M`

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

Token	Description
\$A	Insert data type acronym (for example, <code>i32</code> for integers) into signal and work vector identifiers.
\$M	Insert name mangling string if required to avoid naming collisions. Required.
\$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrBlkIO

Type: string

Value: valid combination of tokens

Default: rtb_\$\$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	rtb_\$\$M

See Also

- “Identifier Format Control” in the Embedded Coder documentation
- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Constant macros

Customize generated constant macro identifiers.

Settings

Default: \$R\$N\$M

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

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

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 a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model,

the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrMacro

Type: string

Value: valid combination of tokens

Default: \$R\$N\$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	\$R\$N\$M

See Also

- “Identifier Format Control” in the Embedded Coder documentation
- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Avoid Identifier Name Collisions with Referenced Models” in the Embedded Coder documentation

- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Shared utilities

Customize shared utility identifiers.

Settings

Default: \$N\$C

Customize generated shared utility identifier names.

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

Token	Description
\$N	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated. Optional.
\$C	Insert eight-character conditional checksum when \$N is not specified or the Maximum identifier length does not accommodate the full length of \$N. Required.

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, the checksum is included in the identifier name. Otherwise, the code generator includes the checksum when necessary to prevent name collisions.
- If you specify \$N, then the checksum is only included in the name when the identifier length is too short to accommodate the fully expanded format string. The code generator includes the checksum and truncates \$N 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.

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrUtil

Type: string

Value: valid combination of tokens

Default: \$N\$C

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	\$N\$C

See Also

- “Code Generation Pane: Symbols” on page 4-102
- “Identifier Format Control”
- “Exceptions to Identifier Formatting Conventions”

Minimum mangle length

Increase the minimum number of characters for generating name mangling strings to help avoid name collisions.

Settings

Default: 1

Specify an integer value that indicates the minimum number of characters the code generator uses when generating a name mangling string. 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 string. The length of the name mangling string increases as the number of name collisions increases.

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: MangleLength

Type: integer

Value: value between 1 and 15

Default: 1

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	1
Efficiency	No impact
Safety precaution	No impact

See Also

- “Control Name Mangling in Generated Identifiers” in the Embedded Coder documentation
- “Maintain Traceability for Generated Identifiers” in the Embedded Coder documentation

Maximum identifier length

Specify maximum number of characters in generated function, type definition, variable names.

Settings

Default: 31

Minimum: 31

Maximum: 256

You can use this parameter to limit the number of characters in function, type definition, and variable names.

Tips

- Consider increasing identifier length for models having a deep hierarchical structure.
- When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate the root model name, and possibly, the name mangling string. A code generation error occurs if **Maximum identifier length** is too small.
- This parameter must be the same for both top-level and referenced models.
- When a name conflict occurs between a symbol within the scope of a higher level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. Name mangling is performed on the symbol from the higher level model.

Command-Line Information

Parameter: MaxIdLength

Type: integer

Value: valid value

Default: 31

Recommended Settings

Application	Setting
Debugging	Valid value
Traceability	>30
Efficiency	No impact
Safety precaution	>30

See Also

- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Identifier Format Control” in the Embedded Coder documentation

System-generated identifiers

Specify whether the code generator uses shorter, more consistent names for the \$N token in system-generated identifiers.

Settings

Default: Shortened

Classic

Generate longer identifier names, which are used by default before R2013a, for the \$N token. For example, for a model named sym, if:

- “Global variables” on page 4-106 is \$N\$R\$M, the block state identifier is sym_DWork.
- “Global types” on page 4-109 is \$R\$N\$M, the block state type is a structure named D_Work_sym.

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” on page 4-106 is \$N\$R\$M, the block state identifier is sym_DW.
- “Global types” on page 4-109 is \$R\$N\$M, the block state type is a structure named DW_sym.

System-generated identifiers per model

Classic	Shortened	Data Representation	Description
BlockIO, B	B	Type	Block signals of the system
ExternalInputs	ExtU	Type	Block input data for root system

System-generated identifiers per model (Continued)

Classic	Shortened	Data Representation	Description
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
Parameters	P	Type	Parameters for the system
ConstBlockIO	ConstB	Const Type	Block inputs and outputs that are constants
MachineLocalData, Machine	MachLocal	Const Type, Global Variable	Used by ERT 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

System-generated identifiers per model (Continued)

Classic	Shortened	Data Representation	Description
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
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
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

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

Dependencies

- This parameter appears only for ERT-based targets.
- When generating code, this parameter requires an Embedded Coder license.

Command-Line Information

Parameter: InternalIdentifier

Type: string

Value: Classic | Shortened

Default: Shortened

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Specify Identifier Length to Avoid Naming Collisions”
- “Specify Reserved Names for Generated Identifiers”
- “Customize Generated Identifier Naming Rules” in the Embedded Coder documentation
- “Identifier Format Control” in the Embedded Coder documentation

Generate scalar inlined parameter as

Control expression of scalar inlined parameter values in the generated code.

Settings

Default: Literals

Literals

Generates scalar inlined parameters as numeric constants. This setting can help with debugging TLC code, as it makes it easy to search for parameter values in the generated code.

Macros

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `InlinedPrmAccess`

Type: string

Value: Literals | Macros

Default: Literals

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Macros
Efficiency	Literals
Safety precaution	No impact

Signal naming

Specify rules for naming signals in generated code.

Settings

Default: None

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 M-function

Uses the MATLAB function specified with the **M-function** parameter to create identifiers for signal names in the 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 M-function** enables **M-function**.
- This parameter must be the same for top-level and referenced models.
- If you give a value to the **Alias** parameter of an `MPT.Signal` or `Simulink.Signal` data object, that value overrides the specification of the **Signal naming** parameter.

Limitation

This parameter does not impact signal names that are specified by an embedded signal object created using the **Code Generation** tab of a **Signal Properties** dialog box. See “Custom Storage Classes Using Embedded Signal Objects” for information about embedded signal objects.

Command-Line Information

Parameter: SignalNamingRule

Type: string

Value: None | UpperCase | LowerCase | Custom

Default: None

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Force upper case
Efficiency	No impact
Safety precaution	No impact

See Also

- “Apply Naming Rules to Identifiers Globally” in the Embedded Coder documentation
- “Programming Scripts and Functions” in the MATLAB documentation

M-function

Specify rule for naming identifiers in generated code.

Settings

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.

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Signal naming**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: SignalNamingFcn

Type: string

Value: MATLAB language file

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Apply Naming Rules to Identifiers Globally” in the Embedded Coder documentation
- “Programming Scripts and Functions” in the MATLAB documentation

Parameter naming

Specify rule for naming parameters in generated code.

Settings

Default: None

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 M-function

Uses the MATLAB function specified with the **M-function** parameter to create identifiers for parameter names in the 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 M-function enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: ParamNamingRule

Type: string

Value: None | UpperCase | LowerCase | Custom

Default: None

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Force upper case
Efficiency	No impact
Safety precaution	No impact

See Also

- “Apply Naming Rules to Identifiers Globally” in the Embedded Coder documentation
- “Programming Scripts and Functions” in the MATLAB documentation

#define naming

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

Settings

Default: None

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 M-function

Uses the MATLAB function specified with the **M-function** parameter to create identifiers for `#define` parameter names in the 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 M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `DefineNamingRule`

Type: `string`

Value: `None | UpperCase | LowerCase | Custom`

Default: `None`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Force upper case
Efficiency	No impact
Safety precaution	No impact

See Also

- “Apply Naming Rules to Identifiers Globally” in the Embedded Coder documentation
- “Programming Scripts and Functions” in the MATLAB documentation

Use the same reserved names as Simulation Target

Specify whether to use the same reserved names as those specified in the **Simulation Target > Symbols** pane.

Settings

Default: Off

- On
Enables using the same reserved names as those specified in the **Simulation Target > Symbols** pane.
- Off
Disables using the same reserved names as those specified in the **Simulation Target > Symbols** pane.

Command-Line Information

Parameter: UseSimReservedNames

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

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.

Settings

Default: {}

This action changes the names of variables or functions in the generated code to avoid name conflicts with identifiers in custom code. Reserved names must be shorter than 256 characters.

Tips

- Do not enter Simulink Coder keywords since these names cannot be changed in the generated code. For a list of keywords to avoid, see “Reserved Keywords”.
- Start each reserved name with a letter or an underscore to prevent error messages.
- Each reserved name must contain only letters, numbers, or underscores.
- Separate the reserved names using commas or spaces.
- You can also specify reserved names by using the command line:

```
config_param_object.set_param('ReservedNameArray',  
{ 'abc', 'xyz' })
```

where *config_param_object* is the object handle to the model settings in the Configuration Parameters dialog box.

Command-Line Information

Parameter: ReservedNameArray

Type: string array

Value: reserved names shorter than 256 characters

Default: {}

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Code Generation Pane: Custom Code

The **Code Generation > Custom Code** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT- or ERT-based target.

Use the same custom code settings as Simulation Target

Include custom C code in generated:

Source file	Source file:
Header file	
Initialize function	
Terminate function	

Include list of additional:

Include directories	Include directories:
Source files	
Libraries	

In this section...

“Code Generation: Custom Code Tab Overview” on page 4-153

“Use the same custom code settings as Simulation Target” on page 4-154

“Use local custom code settings (do not inherit from main model)” on page 4-155

“Source file” on page 4-157

“Header file” on page 4-158

“Initialize function” on page 4-159

“Terminate function” on page 4-160

“Include directories” on page 4-161

“Source files” on page 4-163

“Libraries” on page 4-165

Code Generation: Custom Code Tab Overview

Enter custom code to include in generated model files and create a list of additional folders, source files, and libraries to use when building the model.

Configuration

- 1** Select the type of information to include from the list on the left side of the pane.
- 2** Enter custom code or enter a string to identify a folder, source file, or library.
- 3** Click **Apply**.

See Also

- “Configure Model for External Code Integration”
- “Code Generation Pane: Custom Code” on page 4-150

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.

Settings

Default: Off



On

Enables using the same custom code settings as those in the **Simulation Target > Custom Code** pane.



Off

Disables using the same custom code settings as those in the **Simulation Target > Custom Code** pane.

Command-Line Information

Parameter: RTWUseSimCustomCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Use local custom code settings (do not inherit from main model)

Specify if a library model can use custom code settings that are unique from the main model.

Settings

Default: Off



On

Enables a library model to use custom code settings that are unique from the main model.



Off

Disables a library model from using custom code settings that are unique from the main model.

Dependency

This parameter is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.

Command-Line Information

Parameter: RTWUseLocalCustomCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Source file

Specify custom code to include near the top of the generated model source file.

Settings

Default: ''

The code generator places code near the top of the generated *model.c* or *model.cpp* file, outside of any function.

Command-Line Information

Parameter: CustomSourceCode

Type: string

Value: C code

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Header file

Specify custom code to include near the top of the generated model header file.

Settings

Default: ''

The Simulink Coder software places this code near the top of the generated *model.h* header file. If you are including a header file, in your custom header file add `#ifndef` code. This avoids multiple inclusions. For example, in *rtwtypes.h* the following `#include` guards are added:

```
#ifndef RTW_HEADER_rtwtypes_h_
#define RTW_HEADER_rtwtypes_h_
...
#endif /* RTW_HEADER_rtwtypes_h_ */
```

Command-Line Information

Parameter: CustomHeaderCode

Type: string

Value: C code

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Initialize function

Specify custom code to include in the generated model initialize function.

Settings

Default: ''

The Simulink Coder software places code inside the model's initialize function in the *model.c* or *model.cpp* file.

Command-Line Information

Parameter: CustomInitializer

Type: string

Value: C code

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Terminate function

Specify custom code to include in the generated model terminate function.

Settings

Default: ''

Specify code to appear in the model's generated terminate function in the *model.c* or *model.cpp* file.

Dependency

A terminate function is generated only if you select the **Terminate function required** check box on the **Code Generation > Interface** pane.

Command-Line Information

Parameter: CustomTerminator

Type: string

Value: C code

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Include directories

Specify a list of include folders to add to the include path.

Settings

Default: ''

Enter a space-separated list of include folders to add to the include path when compiling the generated code.

- Specify absolute or relative paths to the folders.
- Relative paths must be relative to the folder containing your model files, not relative to the build folder.
- The order in which you specify the folders is the order in which they are searched for header, source, and library files.

Note If you specify a Windows path string containing one or more spaces, you must enclose the string in double quotes. For example, the second and third path strings in the **Include directories** entry below must be double-quoted:

```
C:\Project "C:\Custom Files" "C:\Library Files"
```

If you set the equivalent command-line parameter `CustomInclude`, each path string containing spaces must be separately double-quoted within the single-quoted third argument string, for example,

```
>> set_param('mymodel', 'CustomInclude', ...  
            'C:\Project "C:\Custom Files" "C:\Library Files"')
```

Command-Line Information

Parameter: `CustomInclude`

Type: string

Value: folder path

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Source files

Specify a list of additional source files to compile and link with the generated code.

Settings

Default: ''

Enter a space-separated list of source files to compile and link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

You can specify just the file name if the file is in the current MATLAB folder or in one of the include folders.

Command-Line Information

Parameter: CustomSource

Type: string

Value: file name

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Libraries

Specify a list of additional libraries to link with the generated code.

Settings

Default: ''

Enter a space-separated list of static library files to link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

You can specify just the file name if the file is in the current MATLAB folder or in one of the include folders.

Command-Line Information

Parameter: CustomLibrary

Type: string

Value: library file name

Default: ''

Recommended Settings

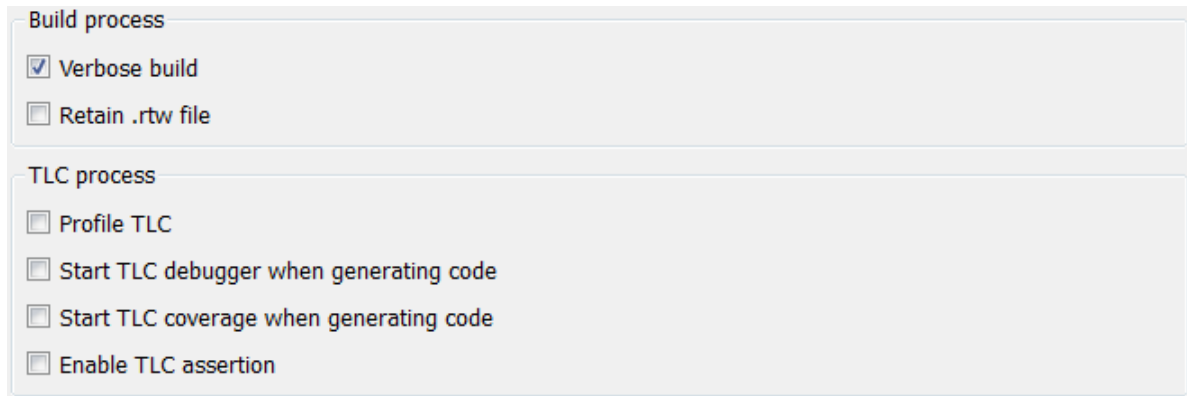
Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure Model for External Code Integration”

Code Generation Pane: Debug

The **Code Generation > Debug** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT- or ERT-based target.



The image shows a screenshot of the 'Code Generation Pane: Debug' settings. It is divided into two sections: 'Build process' and 'TLC process'. The 'Build process' section contains two checkboxes: 'Verbose build' (checked) and 'Retain .rtw file' (unchecked). The 'TLC process' section contains four checkboxes: 'Profile TLC' (unchecked), 'Start TLC debugger when generating code' (unchecked), 'Start TLC coverage when generating code' (unchecked), and 'Enable TLC assertion' (unchecked).

Section	Parameter	Value
Build process	Verbose build	Checked
	Retain .rtw file	Unchecked
TLC process	Profile TLC	Unchecked
	Start TLC debugger when generating code	Unchecked
	Start TLC coverage when generating code	Unchecked
	Enable TLC assertion	Unchecked

In this section...

“Code Generation: Debug Tab Overview” on page 4-169

“Verbose build” on page 4-170

“Retain .rtw file” on page 4-171

“Profile TLC” on page 4-172

“Start TLC debugger when generating code” on page 4-173

“Start TLC coverage when generating code” on page 4-175

“Enable TLC assertion” on page 4-176

Code Generation: Debug Tab Overview

Select build process and Target Language Compiler (TLC) process options.

See Also

- “Debug”
- “Code Generation Pane: Debug” on page 4-167

Verbose build

Display code generation progress.

Settings

Default: on



On

The MATLAB Command Window displays progress information indicating code generation stages and compiler output during code generation.



Off

Does not display progress information.

Command-Line Information

Parameter: RTWVerbose

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	On

See Also

“Debug”

Retain .rtw file

Specify *model*.rtw file retention.

Settings

Default: off



On

Retains the *model*.rtw file in the current build folder. This parameter is useful if you are modifying the target files and need to look at the file.



Off

Deletes the *model*.rtw from the build folder at the end of the build process.

Command-Line Information

Parameter: RetainRTWFile

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Debug”

Profile TLC

Profile the execution time of TLC files.

Settings

Default: off



On

The TLC profiler analyzes the performance of TLC code executed during code generation, and generates an HTML report.



Off

Does not profile the performance.

Command-Line Information

Parameter: ProfileTLC

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Debug”

Start TLC debugger when generating code

Specify use of the TLC debugger

Settings

Default: Off



On

The TLC debugger starts during code generation.



Off

Does not start the TLC debugger.

Tips

- You can also start the TLC debugger by entering the `-dc` argument into the **System target file** field.
- To invoke the debugger and run a debugger script, enter the `-df filename` argument into the **System target file** field.

Command-Line Information

Parameter: TLCDebug

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also
“Debug”

Start TLC coverage when generating code

Generate the TLC execution report.

Settings

Default: off



On

Generates .log files containing the number of times each line of TLC code is executed during code generation.



Off

Does not generate a report.

Tip

You can also generate the TLC execution report by entering the `-dg` argument into the **System target file** field.

Command-Line Information

Parameter: TLCCoverage

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Debug”

Enable TLC assertion

Produce the TLC stack trace

Settings

Default: off



On

The build process halts if a user-supplied TLC file contains an `%assert` directive that evaluates to `FALSE`.



Off

The build process ignores TLC assertion code.

Command-Line Information

Parameter: `TLCAssert`

Type: string

Value: `'on'` | `'off'`

Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	On

See Also

“Debug”

Code Generation Pane: Interface

The **Code Generation > Interface** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

The screenshot shows the 'Code Generation Pane: Interface' settings for a GRT-based target. It is organized into three sections: 'Software environment', 'Code interface', and 'Data exchange'.
- **Software environment:** Includes 'Standard math library' set to 'C89/C90 (ANSI)', 'Code replacement library' set to 'None', and 'Shared code placement' set to 'Auto'. A checked checkbox 'Support non-finite numbers' is also present.
- **Code interface:** Includes 'Code interface packaging' set to 'Nonreusable function' and an unchecked checkbox 'Classic call interface'.
- **Data exchange:** Includes a checked checkbox 'MAT-file logging', 'MAT-file variable name modifier' set to 'rt_', and 'Interface' set to 'None'.

The **Code Generation > Interface** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

Software environment

Standard math library: C89/C90 (ANSI) ▾

Code replacement library: None ▾ Custom...

Shared code placement: Auto ▾

Support: floating-point numbers non-finite numbers complex numbers
 absolute time continuous time non-inlined S-functions
 variable-size signals

Multiword type definitions: System defined ▾

Code interface

Code interface packaging: Nonreusable function ▾

Classic call interface

Single output/update function Terminate function required

Generate preprocessor conditionals: Use local settings ▾

Suppress error status in real-time model data structure Combine signal/state structures

Configure Model Functions

Data exchange

MAT-file logging

Interface: None ▾

In this section...

- “Code Generation: Interface Tab Overview” on page 4-181
- “Standard math library” on page 4-182
- “Code replacement library” on page 4-184
- “Custom” on page 4-187
- “Shared code placement” on page 4-188
- “Support: floating-point numbers” on page 4-190
- “Support: non-finite numbers” on page 4-192
- “Support: complex numbers” on page 4-194
- “Support: absolute time” on page 4-195
- “Support: continuous time” on page 4-197
- “Support: non-inlined S-functions” on page 4-199
- “Support: variable-size signals” on page 4-201
- “Multiword type definitions” on page 4-202
- “Maximum word length” on page 4-204
- “Code interface packaging” on page 4-206
- “Multi-instance code error diagnostic” on page 4-210
- “Pass root-level I/O as” on page 4-212
- “Generate function to allocate model data” on page 4-214
- “Classic call interface” on page 4-216
- “Single output/update function” on page 4-218
- “Terminate function required” on page 4-221
- “Generate preprocessor conditionals” on page 4-223
- “Suppress error status in real-time model data structure” on page 4-225
- “Combine signal/state structures” on page 4-227
- “Configure Model Functions” on page 4-230
- “Block parameter visibility” on page 4-231

In this section...

- “Internal data visibility” on page 4-233
- “Block parameter access” on page 4-235
- “Internal data access” on page 4-237
- “External I/O access” on page 4-239
- “Generate destructor” on page 4-241
- “Use operator new for referenced model object registration” on page 4-243
- “Configure C++ Class Interface” on page 4-245
- “MAT-file logging” on page 4-246
- “MAT-file variable name modifier” on page 4-249
- “Interface” on page 4-251
- “Generate C API for: signals” on page 4-254
- “Generate C API for: parameters” on page 4-255
- “Generate C API for: states” on page 4-256
- “Generate C API for: root-level I/O” on page 4-257
- “Transport layer” on page 4-258
- “MEX-file arguments” on page 4-260
- “Static memory allocation” on page 4-262
- “Static memory buffer size” on page 4-264

Code Generation: Interface Tab Overview

Select the target software environment, output variable name modifier, and data exchange interface.

See Also

- “Specify Target Interfaces”
- “Code Generation Pane: Interface” on page 4-177

Standard math library

Specify a standard math library for your model.

Settings

Default: C89/C90 (ANSI)

C89/C90 (ANSI)

Generates calls to the ISO®/IEC 9899:1990 C standard math library for floating-point functions.

C99 (ISO)

Generates calls to the ISO/IEC 9899:1999 C standard math library.

C++03 (ISO)

Generates calls to the ISO/IEC 14882:2003 C++ standard math library.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Dependencies

The C++03 (ISO) math library is available for use only if you select C++ for the **Language** parameter.

Command-Line Information

Parameter: TargetLangStandard

Type: string

Value: 'C89/C90 (ANSI)' | 'C99 (ISO)' | 'C++03 (ISO)'

Default: 'C89/C90 (ANSI)'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

“Specify Target Interfaces”

Code replacement library

Specify an application-specific math library for your model.

Settings

Default: None

None

Does not use a code replacement library.

GNU C99 extensions

Generates calls to the GNU® gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

Intel IPP

Generates calls to the Intel Performance Primitives (IPP) library.

Intel IPP/SSE with GNU99 extensions

Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions.

- Additional values might be listed for licensed target products and for embedded and desktop targets. If you have created and registered code replacement libraries using the Embedded Coder product, additional values are listed.
- The software filters the list of **Code replacement library** values based on compatibility with the **Language**, **Standard math library**, and **Device vendor** values you select for your model.

Tips

- If you specify Shared location for the **Code Generation > Interface > Shared code placement** parameter or you generate code for models in a model reference hierarchy,
 - Models that are sharing the location or are in the model hierarchy must specify the same code replacement library (same name, tables, and table entries).

- The code generator reports a checksum warning (see “Shared Utility Checksum”) 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 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* specify Shared location for the **Code Generation > Interface > Shared code placement** parameter.
 - You change the code replacement library name or contents before regenerating code.

Tip

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.

Command-Line Information

Parameter: CodeReplacementLibrary

Type: string

Value: 'None' | 'GNU C99 extensions' | 'Intel IPP' | 'Intel IPP/SSE with GNU99 extensions'

Default: 'None'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

“Specify Target Interfaces”

Custom

Open the Code Replacement Tool. With this tool, you can you create and manage the code replacement tables that make up a code replacement library (CRL).

Dependencies

- This button appears only for ERT-based targets.
- This button requires an Embedded Coder license when generating code.

See Also

- “Manage Code Replacement Tables with the Code Replacement Tool”
- “Code Replacement”

Shared code placement

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Settings

Default: Auto

Auto

Operates as follows:

- When the model contains Model blocks, places utility code within the `slprj/target/_sharedutils` folder.
- When the model does not contain Model blocks, places utility code in the build folder (generally, in `model.c` or `model.cpp`).

Shared location

Directs code for utilities to be placed within the `slprj` folder in your working folder.

Command-Line Information

Parameter: UtilityFuncGeneration

Type: string

Value: 'Auto' | 'Shared location'

Default: 'Auto'

Recommended Settings

Application	Setting
Debugging	Shared location (GRT) No impact (ERT)
Traceability	Shared location (GRT) No impact (ERT)
Efficiency	No impact (execution, RAM) Shared location (ROM)
Safety precaution	No impact

See Also

- “Specify Target Interfaces”
- “Sharing Utility Code”

Support: floating-point numbers

Specify whether to generate floating-point data and operations.

Settings

Default: On (GUI), 'off' (command-line)



On

Generates floating-point data and operations.



Off

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

Dependencies

- This option only appears for ERT-based targets.
- This option requires an Embedded Coder license when generating code.
- Selecting this option enables **Support: non-finite numbers** and clearing this option disables **Support: non-finite numbers**.
- This option must be the same for top-level and referenced models.

Command-Line Information

Parameter: PurelyIntegerCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Note The command-line values are reverse of the settings values. Therefore, 'on' in the command line corresponds to the description of “Off” in the settings section, and 'off' in the command line corresponds to the description of “On” in the settings section.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (GUI), 'on' (command-line) — for integer only
Safety precaution	No impact

Support: non-finite numbers

Specify whether to generate nonfinite data and operations on nonfinite data.

Settings

Default: on

- On
Generates nonfinite data (for example, NaN and Inf) and related operations.
- Off
Does not generate nonfinite data and operations. If you clear this option, an error occurs if the code generator encounters nonfinite data or expressions. The error message reports offending blocks and parameters.

Note Code generation is optimized with the assumption that nonfinite data are absent. However, if your application produces nonfinite numbers through signal data or MATLAB code, the behavior of the generated code might be inconsistent with simulation results when processing nonfinite data.

Dependencies

- For ERT-based targets, this parameter is enabled by **Support: floating-point numbers**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: SupportNonFinite
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

Support: complex numbers

Specify whether to generate complex data and operations.

Settings

Default: on

On
Generates complex numbers and related operations.

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: SupportComplex
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (for real only)
Safety precaution	No impact

Support: absolute time

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

Settings

Default: on



On

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

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



Off

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- You must select this parameter if your model includes blocks that require absolute or elapsed time values.

Command-Line Information

Parameter: SupportAbsoluteTime

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

“Timers”

Support: continuous time

Specify whether to generate code for blocks that use continuous time.

Settings

Default: off



On

Generates code for blocks that use continuous time.



Off

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

Dependencies

- This option only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This option must be on if your model includes blocks that require absolute or elapsed time values.
- If you have customized `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, be aware that selecting the options **Support: continuous time** and **Single output/update function** together may cause output values read from `ert_main` for a continuous output port to differ from the corresponding output values in the logged data for the model. This is because, while logged data is a snapshot of output at major time steps, output read from `ert_main` after the base-rate model step potentially reflects intervening minor time steps. To work around this limitation and eliminate the discrepancy, do one of the following:
 - Separate the generated output and update functions (clear the **Single output/update function** option), and insert code in `ert_main` to read model output values reflecting only the major time steps. For example, in `ert_main`, between the `model_output` call and the `model_update` call, read the model External outputs global data structure (defined in `model.h`).

- If you want to keep the **Single output/update function** option selected, insert code in the generated `model.c` or `.cpp` file to return model output values reflecting only the major time steps. For example, in the model step function, between the output code and the update code, you could save the value of the model `External outputs` global data structure (defined in `model.h`), and then restore the value after the update code completes.
- Place a Zero-Order Hold block before the continuous output port.

Command-Line Information

Parameter: SupportContinuousTime

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

“Use Discrete and Continuous Time”

Support: non-inlined S-functions

Specify whether to generate code for noninlined S-functions.

Settings

Default: Off



On

Generates code for noninlined S-functions.



Off

Does not generate code for noninlined S-functions. If this parameter is off and the model includes a noninlined S-function, an error occurs 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.
- Noninlined 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.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Selecting this parameter also selects **Support: floating-point numbers** and **Support: non-finite numbers**. If you clear **Support: floating-point numbers** or **Support: non-finite numbers**, a warning is displayed during code generation because these parameters are required by the S-function interface.

Command-Line Information

Parameter: SupportNonInlinedSFcns

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

“Insert S-Function Code”

Support: variable-size signals

Specify whether to generate code for models that use variable-size signals.

Settings

Default: Off



On

Generates code for models that use variable-size signals.



Off

Does not generate code for models that use variable-size signals. If this parameter is off and the model uses variable-size signals, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: SupportVariableSizeSignals

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	Off

Multiword type definitions

Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.

Settings

Default: System defined

System defined

Use the default system type definitions for multiword data types in generated code. During code generation, if multiword usage is detected, multiword type definitions are generated into the file `multiword_types.h`.

User defined

Allows you to control how multiword type definitions are handled during the code generation process. Selecting this value enables the associated parameter **Maximum word length**, which allows you to specify a maximum word length, in bits, for which the code generation process generates multiword type definitions into the file `multiword_types.h`. The default maximum word length is 256. If you select 0, multiword type definitions are not generated into 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 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.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Selecting the value `User` defined for this parameter enables the associated parameter **Maximum word length**.

Command-Line Information

Parameter: `ERTMultiwordTypeDef`

Type: `string`

Value: `'System defined' | 'User defined'`

Default: `'System defined'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	Use default

Maximum word length

Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.

Settings

Default: 256

Specify a maximum word length, in bits, for which the code generation process generates multiword type definitions into the file `multiword_types.h`. All multiword type definitions up to and including this number of bits are generated. If you select 0, multiword type definitions are not generated into 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 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.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by selecting the value `User` defined for the parameter **Multiword type definitions**.

Command-Line Information

Parameter: ERTMultiwordLength

Type: integer

Value: valid quantity of bits representing a word size

Default: 256

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	Use default

Code interface packaging

Select the packaging for the generated C or C++ code interface.

Settings

Default: Nonreusable function if **Language** is set to C; C++ class if **Language** is set to C++

C++ class

Generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.

Nonreusable function

Generate nonreusable code. Model data structures are statically allocated and accessed by model entry point functions directly in the model code.

Reusable function

Generate reusable, multi-instance code that is reentrant, as follows:

- For a GRT-based model, the generated *model.c* source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, you can use the **Generate function to allocate model data** option to control whether an allocation function is generated.
- The generated code passes the real-time model data structure in, by reference, as an argument to *model_step* and the other model entry point functions.
- The real-time model data structure is exported with the *model.h* header file.

For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the reusable model entry-point functions. They can be included in the real-time model data structure that is passed to the functions, passed as individual arguments, or passed as references to an input structure and an output structure.

Tips

- Entry points are exported with *model.h*. To call the entry-point functions from hand-written code, add an `#include model.h` directive to the code.
- When `Reusable` function is selected, the code generator generates a pointer to the real-time model object (*model_M*).
- In some cases, when `Reusable` function is selected, the code generator might generate code that compiles but is not reentrant. For example, if a signal, `DWork` structure, or parameter data has a storage class other than `Auto`, global data structures are generated.

Dependencies

- The value `C++ class` is available only if the **Language** parameter is set to `C++` on the **Code Generation** pane.
- Selecting `Reusable` function or `C++ class` enables **Multi-instance code error diagnostic**.
- For an ERT target, selecting `Reusable` function enables **Pass root-level I/O as** and **Generate function to allocate model data**.
- For an ERT target, selecting `C++ class` enables the following controls for customizing the model class interface:
 - **Configure C++ Class Interface** button
 - **Data Member Visibility/Access Control** subpane
 - Model options **Generate destructor** and **Use operator new for referenced model object registration**
- For an ERT target, you can use `Reusable` function with the static `ert_main.c` module, provided that you do the following:
 - Select the value `Part of model data structure` for **Pass root-level I/O as**.
 - Select the option **Generate function to allocate model data**.
- For an ERT target, you cannot use `Reusable` function if you are using:
 - The `model_step` function prototype control capability

- The subsystem parameter **Function with separate data**
- A subsystem that
 - Has multiple ports that share the same source
 - Has a port that is used by multiple instances of the subsystem and has different sample times, data types, complexity, frame status, or dimensions across the instances
 - Has output marked as a global signal
 - For each instance contains identical blocks with different names or parameter settings
- Using `Reusable` function does not impact the code generated for function-call subsystems.

Command-Line Information

Parameter: `CodeInterfacePackaging`

Type: string

Value: 'C++ class' | 'Nonreusable function' | 'Reusable function'

Default: 'Nonreusable function' if `TargetLang` is set to 'C'; 'C++ class' if `TargetLang` is set to 'C++'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Reusable function or C++ class
Safety precaution	No impact

See Also

- “Entry-Point Functions and Scheduling”
- “Generate Reentrant Code from Top-Level Models” (GRT)

- “Use GRT with Reusable Function Packaging to Combine Models”
- “Generate Reentrant Code from Top-Level Models” (ERT)
- “Generate C++ Class Interface to Model or Subsystem Code” (GRT)
- “C++ Class Interface Control” (ERT)
- “Code Generation of Subsystems”
- “Code Reuse Limitations for Subsystems”
- “Determine Why Subsystem Code Is Not Reused”
- “S-Functions That Support Code Reuse”
- “Static Main Program Module”
- “Function Prototype Control”
- “Atomic Subsystem Code”
- “Export Function-Call Subsystems”
- `model_step`

Multi-instance code error diagnostic

Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.

Settings

Default: Error

None

Proceed with build without displaying a diagnostic message.

Warning

Proceed with build after displaying a warning message.

Error

Abort build after displaying an error message.

Under certain conditions, the software might

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

Dependencies

This parameter is enabled by setting **Code interface packaging** to Reusable function or C++ class.

Command-Line Information

Parameter: MultiInstanceErrorCode

Type: string

Value: 'None' | 'Warning' | 'Error'

Default: 'Error'

Recommended Settings

Application	Setting
Debugging	Warning or Error
Traceability	No impact
Efficiency	None
Safety precaution	No impact

See Also

- “Entry-Point Functions and Scheduling”
- “Generate Reentrant Code from Top-Level Models”
- “Generate C++ Class Interface to Model or Subsystem Code”
- “Code Generation of Subsystems”
- “Code Reuse Limitations for Subsystems”
- “Determine Why Subsystem Code Is Not Reused”
- “Atomic Subsystem Code”

Pass root-level I/O as

Control how root-level model input and output are passed to the reusable `model_step` function.

Settings

Default: Individual arguments

Individual arguments

Passes each root-level model input and output value to `model_step` as a separate argument.

Structure reference

Packs root-level model input into a struct and passes struct to `model_step` as an argument. Similarly, packs root-level model output into a second struct and passes it to `model_step`.

Part of model data structure

Packages root-level model input and output into the real-time model data structure.

Dependencies

- This parameter only appears for ERT-based targets with **Code interface packaging** set to Resuable function.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: RootIOFormat

Type: string

Value: 'Individual arguments' | 'Structure reference' | 'Part of model data structure'

Default: 'Individual arguments'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Entry-Point Functions and Scheduling”
- “Generate Reentrant Code from Top-Level Models”
- “Code Generation of Subsystems”
- “Atomic Subsystem Code”
- `model_step`

Generate function to allocate model data

Control how the generated code allocates memory for model data.

Settings

Default: off



On

Generates a function to dynamically allocate memory (using `malloc`) for model data structures.



Off

Does not generate a dynamic memory allocation function. The generated code statically allocates memory for model data structures.

Dependencies

- This parameter only appears for ERT-based targets with **Code interface packaging** set to `Reusable` function.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `GenerateAllocFcn`

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	Off

See Also

- “Entry-Point Functions and Scheduling”
- “Generate Reentrant Code from Top-Level Models”
- “Code Generation of Subsystems”
- “Atomic Subsystem Code”
- `model_step`

Classic call interface

Specify whether to generate model function calls compatible with the main program module of the GRT target in models created before R2012a.

Settings

Default: off (except on for GRT models created before R2012a)



On

Generates model function calls that are compatible with the main program module of the GRT target (`grt_main.c` or `grt_main.cpp`) in models created before R2012a.

This option provides a quick way to use code generated in the current release with a GRT-based custom target that has a main program module based on pre-R2012a `grt_main.c` or `grt_main.cpp`.



Off

Disables the classic call interface.

Tips

The following are unsupported:

- Data type replacement
- Nonvirtual subsystem option **Function with separate data**

Dependencies

- For an ERT target, setting **Code interface packaging** to **C++ class** disables this option.
- For an ERT target, selecting this option also selects the required option **Support: floating-point numbers**. If you subsequently clear **Support: floating-point numbers**, an error is displayed during code generation.
- For an ERT target, selecting this option disables the incompatible option **Single output/update function**. Clearing this option enables (but does not select) **Single output/update function**.

Command-Line Information

Parameter: GRTInterface

Type: string

Value: 'on' | 'off'

Default: 'off' (except 'on' for GRT models created before R2012a)

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

“Use Discrete and Continuous Time”

Single output/update function

Specify whether to generate the *model_step* function.

Settings

Default: on

On

Generates the *model_step* function for a model. This function contains the output and update function code for the blocks in the model and is called by `rt_OneStep` to execute processing for one clock period of the model at interrupt level.

Off

Does not combine output and update function code into a single function, and instead generates the code in separate *model_output* and *model_update* functions.

Tips

Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. See “Model Blocks and Direct Feed through” for details.

Simulink Coder ignores this parameter for a referenced model if any of the following conditions apply to that model:

- Is multi-rate
- Has a continuous sample time
- Is logging states (using the **States** or **Final states** parameters in the **Configuration Parameters > Data Import/Export** pane)

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting **Code interface packaging** to `C++ class` disables this option.

- This option and **Classic call interface** are mutually incompatible and cannot both be selected through the GUI. Selecting **Classic call interface** disables this option and clearing **Classic call interface** enables this option.
- When you use this option, you must clear the option **Minimize algebraic loop occurrences** on the **Model Referencing** pane.
- If you have customized `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, be aware that selecting the options **Support: continuous time** and **Single output/update function** together may cause output values read from `ert_main` for a continuous output port to differ from the corresponding output values in the logged data for the model. This is because, while logged data is a snapshot of output at major time steps, output read from `ert_main` after the base-rate model step potentially reflects intervening minor time steps. To work around this limitation and eliminate the discrepancy, do one of the following:
 - Separate the generated output and update functions (clear the **Single output/update function** option), and insert code in `ert_main` to read model output values reflecting only the major time steps. For example, in `ert_main`, between the `model_output` call and the `model_update` call, read the model `External outputs` global data structure (defined in `model.h`).
 - If you want to keep the **Single output/update function** option selected, insert code in the generated `model.c` or `.cpp` file to return model output values reflecting only the major time steps. For example, in the model step function, between the output code and the update code, you could save the value of the model `External outputs` global data structure (defined in `model.h`), and then restore the value after the update code completes.
 - Place a Zero-Order Hold block before the continuous output port.

Command-Line Information

Parameter: CombineOutputUpdateFcns

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	On
Safety precaution	On

See Also

“rt_OneStep and Scheduling Considerations”

Terminate function required

Specify whether to generate the `model_terminate` function.

Settings

Default: on



On

Generates a `model_terminate` function. This function contains model termination code and should be called as part of system shutdown.



Off

Does not generate a `model_terminate` function. Suppresses the generation of this function if you designed your application to run indefinitely and does not require a terminate function.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `IncludeMdlTerminateFcn`

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

`model_terminate`

Generate preprocessor conditionals

Generate preprocessor conditional directives globally for a model or locally for each Model block with variant models.

Settings

Default: Use local settings

Use local settings

Generates preprocessor conditional directives based on the value of the **Generate preprocessor conditionals** parameter on the Model block parameters dialog. If you select the **Generate preprocessor conditionals** parameter in the Model block parameters dialog, the generated code contains preprocessor conditional directives for all variant models of that Model block. If you do not select this parameter for a Model block, code is generated for the active variant model.

Enable all

Generates preprocessor conditional directives for all variant models of the Model blocks. Disables the **Generate preprocessor conditionals** option in the Model block parameters dialog.

Disable all

Only generates code for the active variant model of the Model block. Disables the **Generate preprocessor conditionals** option in the Model block parameters dialog for Model blocks.

Tips

For generating preprocessor directives we recommend the following settings:

- Select the “Inline parameters” parameter on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.
- Deselect the “Ignore custom storage classes” on page 4-28 parameter on the **Code Generation** pane of the Configuration Parameters dialog box.

Dependencies

- This parameter only appears for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Use local settings` enables **Generate preprocessor conditionals** parameter on the Model block parameters dialog.
- Setting this parameter to `Enable all` or `Disable all` disables the **Generate preprocessor conditionals** check box on the Model block parameters dialog.
- Setting this parameter to `Enable all` sets the **Selected variant** control on the Model block parameter dialog to `(derive from conditions)`.

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`

Type: `string`

Value: `'Use local settings' | 'Enable all' | 'Disable all'`

Default: `'Use local settings'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Workflow for Implementing Variants”
- “Variant Systems”

Suppress error status in real-time model data structure

Specify whether to log or monitor error status.

Settings

Default: off



On

Omits the error status field from the generated real-time model data structure `rtModel`. This option reduces memory usage.

Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.



Off

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is cleared if you select the incompatible option **MAT-file logging**. If you subsequently select this parameter, code generation displays an error.
- Selecting this parameter clears **Support: continuous time**.
- If your application contains multiple integrated models, the setting of this option must be the same for all of the models to avoid unexpected application behavior. For example, if you select the option for one model but not another, an error status might not get registered by the integrated application.

Command-Line Information

Parameter: SuppressErrorStatus

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	On

See Also

“Use the Real-Time Model Data Structure”

Combine signal/state structures

Specify whether to combine global block signals and global state data into one data structure in the generated code

Settings

Default: Off

On

Combine global block signal data (block I/O) and global state data (DWork vectors) into one data structure in the generated code.

Off

Store global block signals and global states in separate data structures, block I/O and DWork vectors, in the generated code.

Tips

The benefits to setting this parameter to *On* 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 reference block functions, which reduces stack usage
- Better readable data structures with more consistent element sorting

Example. 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;
```

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;
```

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: CombineSignalStateStructs

Type: string

Value: 'on' | 'off'

Default: off

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

- “Global Block I/O Structure”
- “State Storage”

Configure Model Functions

Open the Model Interface dialog box. In this dialog box, you can specify whether the code generator uses default `model_initialize` and `model_step` function prototypes or model-specific C prototypes. Based on your selection, you can preview and modify the function prototypes.

Dependencies

- This button appears only for ERT-based targets with **Code interface packaging** set to a value other than `C++ class`.
- This button requires an Embedded Coder license when generating code.
- This button is active only if your model uses an attached configuration set. If your model uses a referenced configuration set, the button is greyed out. If you want to configure a model-specific step function prototype for a referenced configuration set, use the MATLAB function prototype control functions described in “Configure Function Prototypes Programmatically”.

See Also

- “Function Prototype Control”
- `model_initialize`
- `model_step`
- “Launch the Model Interface Dialog Boxes”

Block parameter visibility

Specify whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class.

Settings

Default: `private`

`public`

Generates the block parameter structure as a `public` data member of the C++ model class.

`private`

Generates the block parameter structure as a `private` data member of the C++ model class.

`protected`

Generates the block parameter structure as a `protected` data member of the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `ParameterMemberVisibility`

Type: `string`

Value: `'public' | 'private' | 'protected'`

Default: `'private'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	protected

See Also

“Configure Code Interface Options”

Internal data visibility

Specify whether to generate internal data structures such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states as public, private, or protected data members of the C++ model class.

Settings

Default: private

public

Generates internal data structures as public data members of the C++ model class.

private

Generates internal data structures as private data members of the C++ model class.

protected

Generates internal data structures as protected data members of the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: InternalMemberVisibility

Type: string

Value: 'public' | 'private' | 'protected'

Default: 'private'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	protected

See Also

“Configure Code Interface Options”

Block parameter access

Specify whether to generate access methods for block parameters for the C++ model class.

Settings

Default: None

None

Does not generate access methods for block parameters for the C++ model class.

Method

Generates noninlined access methods for block parameters for the C++ model class.

Inlined method

Generates inlined access methods for block parameters for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateParameterAccessMethods

Type: string

Value: 'None' | 'Method' | 'Inlined method'

Default: 'None'

Recommended Settings

Application	Setting
Debugging	Inlined method
Traceability	Inlined method

Application	Setting
Efficiency	Inlined method
Safety precaution	None

See Also

“Configure Code Interface Options”

Internal data access

Specify whether to generate access methods for internal data structures, such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states, for the C++ model class.

Settings

Default: None

None

Does not generate access methods for internal data structures for the C++ model class.

Method

Generates noninlined access methods for internal data structures for the C++ model class.

Inlined method

Generates inlined access methods for internal data structures for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateInternalMemberAccessMethods

Type: string

Value: 'None' | 'Method' | 'Inlined method'

Default: 'None'

Recommended Settings

Application	Setting
Debugging	Inlined method
Traceability	Inlined method
Efficiency	Inlined method
Safety precaution	None

See Also

“Configure Code Interface Options”

External I/O access

Specify whether to generate access methods for root-level I/O signals for the C++ model class.

Note This parameter affects generated code only if you are using the default (void-void style) step method for your model class; *not* if you are explicitly passing arguments for root-level I/O signals using an I/O arguments style step method. For more information, see “Passing No Arguments (void-void)” and “Passing I/O Arguments”.

Settings

Default: None

None

Does not generate access methods for root-level I/O signals for the C++ model class.

Method

Generates noninlined access methods for root-level I/O signals for the C++ model class. The software generates set and get methods for each signal.

Inlined method

Generates inlined access methods for root-level I/O signals for the C++ model class. The software generates set and get methods for each signal.

Structure-based method

Generates noninlined, structure-based access methods for root-level I/O signals for the C++ model class. The software generates one set method, taking the address of the external input structure as an argument, and for external outputs (if applicable), one get method, returning the reference to the external output structure.

Inlined structure-based method

Generates inlined, structure-based access methods for root-level I/O signals for the C++ model class. The software generates one set method, taking the address of the external input structure as an argument, and for external outputs (if applicable), one get method, returning the reference to the external output structure.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateExternalIOAccessMethods

Type: string

Value: 'None' | 'Method' | 'Inlined method' | 'Structure-based method' | 'Inlined structure-based method'

Default: 'None'

Recommended Settings

Application	Setting
Debugging	Inlined method
Traceability	Inlined method
Efficiency	Inlined method
Safety precaution	None

See Also

“Configure Code Interface Options”

Generate destructor

Specify whether to generate a destructor for the C++ model class.

Settings

Default: on



On

Generates a destructor for the C++ model class.



Off

Does not generate a destructor for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateDestructor

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	Off

See Also

“Configure Code Interface Options”

Use operator new for referenced model object registration

Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ class interface.

Settings

Default: off



On

Generates code that uses dynamic memory allocation to instantiate objects for referenced models configured with a C++ class interface. Specifically, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses `new` to instantiate objects for referenced models.

Selecting this option frees a parent model from having to maintain information about referenced models beyond its direct children.

- If you select this option, be aware that a `bad_alloc` exception might be thrown, per the C++ standard, if an out-of-memory error occurs during the use of `new`. 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 a top model object.
- If **Use operator new for referenced model object registration** is selected 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. The purpose of the functions is to prevent pointer members within the model class from being copied by other code. For more information, see “Model Class Copy Constructor and Assignment Operator”.



Off

Does not generate code that uses `new` to instantiate referenced model objects.

Clearing this option means that a parent model maintains information about its referenced models, including its direct and indirect children.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: UseOperatorNewForModelRefRegistration

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	Off

See Also

“Configure Code Interface Options”

Configure C++ Class Interface

Open the Configure C++ class interface dialog box. In this dialog box, you can customize the C++ class interface for your model code. Based on your selections, you can preview and modify the model-specific C++ class interface.

Dependencies

- This button appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This button requires an Embedded Coder license when generating code.
- This button is active only if your model uses an attached configuration set. If your model uses a referenced configuration set, the button is greyed out. If you want to configure a model-specific C++ class interface for a referenced configuration set, use the MATLAB C++ class interface control functions described in “Customize C++ Class Interfaces Programmatically”.

See Also

- “C++ Class Interface Control”
- `model_step`
- “Configure Step Method for Your Model Class”

MAT-file logging

Specify MAT-file logging

Settings

Default: on for the GRT target, off for ERT-based targets



On

Enable MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in one of the following ways:

- **Configuration Parameters > Data Import/Export, Save to workspace** subpane (see “Data Import/Export Pane”)
- To Workspace blocks
- To File blocks
- Scope blocks with the **Save data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in “Export Simulation Data” and “Configure Signal Data for Logging”. Setting MAT-file logging redirects the data to a MAT-file instead. The file is named *model.mat*, where *model* is the name of your model.



Off

Disable MAT-file logging. Clearing this option has the following benefits:

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

Dependencies

- For the GRT target, selecting this option also selects the required option **Support non-finite numbers**. If you subsequently clear **Support non-finite numbers**, an error is displayed during code generation.
- For ERT-based targets, selecting this option also selects the required options **Support: floating-point numbers**, **Support: non-finite numbers**, and **Terminate function required**. If you subsequently clear **Support: floating-point numbers**, **Support: non-finite numbers**, or **Terminate function required**, an error is displayed during code generation.
- For ERT-based targets, selecting this option clears the incompatible option **Suppress error status in real-time model data structure**. If you subsequently select **Suppress error status in real-time model data structure**, an error is displayed during code generation.
- Selecting this option enables **MAT-file variable name modifier**.
- For ERT-based targets, clear this option if you are using exported function calls.
- For ERT-based targets, clear this option if you are using **Enable portable word sizes** on the **Code Generation > Verification** pane.

Limitations

In a referenced model, only the following data logging features are supported:

- To File blocks
- State logging — the software stores the data in the MAT-file for the top model.

In the context of the Embedded Coder product, MAT-file logging does not support the following IDEs: Analog Devices™ VisualDSP++, Green Hills® MULTI, IAR Embedded Workbench, Texas Instruments Code Composer Studio, Wind River DIAB/GCC.

Command-Line Information

Parameter: MatFileLogging

Type: string

Value: 'on' | 'off'

Default: 'on' for the GRT target, 'off' for ERT-based targets

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

- “Logging”
- “Log Data for Analysis”
- “Virtualized Output Ports Optimization”

MAT-file variable name modifier

Select the string to add to MAT-file variable names.

Settings

Default: `rt_`

`rt_` Adds a prefix string.

`_rt` Adds a suffix string.

`none` Does not add a string.

Dependency

If you have an Embedded Coder license, for the GRT target or ERT-based targets, this parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: `LogVarNameModifier`

Type: `string`

Value: `'none' | 'rt_' | '_rt'`

Default: `'rt_'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Logging”
- “Log Data for Analysis”

Interface

Specify a data exchange interface to include in the generated code.

Settings

Default: None

None

Does not generate extra code to support a data exchange interface.

C API

Generates code for the C API data interface.

External mode

Generates code for the External mode data interface.

ASAP2

Generates code for the ASAP2 data interface.

Dependencies

Selecting **C API** enables the following parameters:

- **Generate C API for: signals**
- **Generate C API for: parameters**
- **Generate C API for: states**
- **Generate C API for: root-level I/O**

Selecting **External mode** enables the following parameters:

- **Transport layer**
- **MEX-file arguments**
- **Static memory allocation**

Command-Line Information

Parameter: see table

Type: string

Value: 'on' | 'off'

Default: 'off'

To enable...	Set this parameter...	To this value...
None	RTWCAPIParams, RTWCAPISignals, RTWCAPIStates, RTWCAPIRootIO, ExtMode, GenerateASAP2	'off'
C API	RTWCAPIParams, RTWCAPISignals, RTWCAPIStates, RTWCAPIRootIO	'on'
External mode	ExtMode	'on'
ASAP2	GenerateASAP2	'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development None for production code generation

See Also

- “Data Interchange Using the C API”
- “Host/Target Communication”

- “ASAP2 Data Measurement and Calibration”

Generate C API for: signals

Generate a C API signals structure.

Settings

Default: on

- On
Generates C API interface to global block outputs.
- Off
Does not generate C API signals.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information

Parameter: RTWCAPISignals

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Data Interchange Using the C API”

Generate C API for: parameters

Generate C API parameter tuning structures.

Settings

Default: on



On

Generates C API interface to global block parameters.



Off

Does not generate C API parameters.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information

Parameter: RTWCAPIParams

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Data Interchange Using the C API”

Generate C API for: states

Generate a C API states structure.

Settings

Default: off

- On
Generates C API interface to discrete and continuous states.
- Off
Does not generate C API states.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information

Parameter: RTWCAPIStates

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Data Interchange Using the C API”

Generate C API for: root-level I/O

Generate a C API root-level I/O structure.

Settings

Default: off



On

Generates a C API interface to root-level inputs and outputs.



Off

Does not generate a C API interface to root-level inputs and outputs.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information

Parameter: RTWCAPIRootIO

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Data Interchange Using the C API”

Transport layer

Specify the transport protocol for communications.

Settings

Default: tcpip

tcpip

Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

serial

Applies a serial transport mechanism. The MEX-file name is `ext_serial_win32_comm`.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. The value is specified either in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`, for targets provided by MathWorks, or in an `sl_customization.m` file, for custom targets and/or custom transports.

Dependency

This parameter is enabled by selecting `External` mode in the **Interface** parameter.

Command-Line Information

Parameter: ExtModeTransport

Type: integer

Value: 0 for TCP/IP | 1 for serial

Default: 0

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Target Interfacing”
- “Create a Transport Layer for External Communication”

MEX-file arguments

Specify arguments to pass to an External mode interface MEX-file for communicating with executing targets.

Settings

Default: ''

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target (for example, 'myputer' or '148.27.151.12')
- Verbosity level (0 for no information or 1 for detailed information)
- TCP/IP server port number (an integer value between 256 and 65535, with a default of 17725)

For a serial transport, `ext_serial_win32_comm` allows three optional arguments:

- Verbosity level (0 for no information or 1 for detailed information)
- Serial port ID (for example, 1 for COM1, and so on)
- Baud rate (selected from the set 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, with a default baud rate of 57600)

Dependency

Depending on the specified “System target file” on page 4-6, this parameter is enabled by the value selection **Data exchange > Interface > External mode** or by an **External mode** check box.

Command-Line Information

Parameter: ExtModeMexArgs

Type: string

Value: valid arguments

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Target Interfacing”
- “Choose Communication Protocol for Client and Server”

Static memory allocation

Control memory buffer for External mode communication.

Settings

Default: off



On

Enables the **Static memory buffer size** parameter for allocating dynamic memory.



Off

Uses a static memory buffer for External mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- Depending on the specified “System target file” on page 4-6, this parameter is enabled by the value selection **Data exchange > Interface > External mode** or by an **External mode** check box.
- This parameter enables **Static memory buffer size**.

Command-Line Information

Parameter: ExtModeStaticAlloc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure External Mode Options for Code Generation”

Static memory buffer size

Specify the memory buffer size for External mode communication.

Settings

Default: 1000000

Enter the number of bytes to preallocate for External mode communications buffers in the target.

Tips

- If you enter too small a value for your application, External mode issues an out-of-memory error.
- To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information

Parameter: ExtModeStaticAllocSize

Type: integer

Value: valid value

Default: 1000000

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure External Mode Options for Code Generation”

Code Generation Pane: RSim Target

The **Code Generation > RSim Target** pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `rsim.tlc` system target file.

The screenshot displays the configuration parameters for the RSim target, organized into three sections:

- Parameter loading:** Contains a checked checkbox labeled "Enable RSim executable to load parameters from a MAT-file".
- Solver:** Features a dropdown menu labeled "Solver selection:" with the value "auto" selected.
- Storage classes:** Contains a checked checkbox labeled "Force storage classes to AUTO".

In this section...

“Code Generation: RSim Target Tab Overview” on page 4-268

“Enable RSim executable to load parameters from a MAT-file” on page 4-269

“Solver selection” on page 4-270

“Force storage classes to AUTO” on page 4-271

Code Generation: RSim Target Tab Overview

Set configuration parameters for rapid simulation.

Configuration

This tab appears only if you specify `rsim.tlc` as the “System target file” on page 4-6.

See Also

- “Configure and Build Model for Rapid Simulation”
- “Run Rapid Simulations”
- “Code Generation Pane: RSim Target” on page 4-266

Enable RSim executable to load parameters from a MAT-file

Specify whether to load RSim parameters from a MAT-file.

Settings

Default: on



On

Enables RSim to load parameters from a MAT-file.



Off

Disables RSim from loading parameters from a MAT-file.

Command-Line Information

Parameter: RSIM_PARAMETER_LOADING

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Create a MAT-File That Includes a Model Parameter Structure”

Solver selection

Instruct the target how to select the solver.

Settings

Default: auto

auto

Lets the target choose the solver. The target uses the Simulink solver module if you specify a variable-step solver on the Solver pane. Otherwise, the target uses a Simulink Coder built-in solver.

Use Simulink solver module

Instructs the target to use the variable-step solver that you specify on the **Solver** pane.

Use fixed-step solvers

Instructs the target to use the fixed-step solver that you specify on the **Solver** pane.

Command-Line Information

Parameter: RSIM_SOLVER_SELECTION

Type: string

Value: 'auto' | 'usesolvermodule' | 'usefixstep'

Default: 'auto'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Force storage classes to AUTO

Specify whether to retain your storage class settings in a model or to use the automatic settings.

Settings

Default: on



On

Forces the Simulink software to determine storage classes.



Off

Causes the model to retain storage class settings.

Tips

- Turn this parameter on for flexible custom code interfacing.
- Turn this parameter off to retain storage class settings such as ExportedGlobal or ImportExtern.

Command-Line Information

Parameter: RSIM_STORAGE_CLASS_AUTO

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Code Generation Pane: S-Function Target

The **Code Generation > S-Function Target** pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `rtwsfcn.tlc` system target file.

- Create new model
- Use value for tunable parameters
- Include custom source code

In this section...

“Code Generation S-Function Target Tab Overview” on page 4-274

“Create new model” on page 4-275

“Use value for tunable parameters” on page 4-276

“Include custom source code” on page 4-277

Code Generation S-Function Target Tab Overview

Control code generated for the S-function target (`rtwsfcn.tlc`).

Configuration

This tab appears only if you specify the S-function target (`rtwsfcn.tlc`) as the “System target file” on page 4-6.

See Also

- “Generated S-Function Block”
- “Code Generation Pane: S-Function Target” on page 4-272

Create new model

Create a new model containing the generated S-function block.

Settings

Default: on



On

Creates a new model, separate from the current model, containing the generated S-function block.



Off

Generates code but a new model is not created.

Command-Line Information

Parameter: CreateModel

Type: string

Value: 'on' | 'off'

Default: 'on'

See Also

“Generated S-Function Block”

Use value for tunable parameters

Use the variable value instead of the variable name in generated block mask edit fields for tunable parameters.

Settings

Default: off



On

Uses variable values for tunable parameters instead of the variable name in the generated block mask edit fields.



Off

Uses variable names for tunable parameters in the generated block mask edit fields.

Command-Line Information

Parameter: UseParamValues

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

“Generated S-Function Block”

Include custom source code

Include custom source code in the code generated for the S-function.

Settings

Default: off



On

Include provided custom source code in the code generated for the S-function.



Off

Do not include custom source code in the code generated for the S-function.

Command-Line Information

Parameter: AlwaysIncludeCustomSrc

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

“Generated S-Function Block”

Code Generation Pane: Tornado Target

The **Code Generation > Tornado Target** pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `tornado.tlc` system target file.

The screenshot shows a configuration window for the Tornado target in Simulink Coder. It is organized into several sections:

- Software environment**:
 - Standard math library: C89/C90 (ANSI) (dropdown)
 - Code replacement library: None (dropdown)
 - Shared code placement: Auto (dropdown)
- Tornado**:
 - MAT-file Logging
 - Code Format: RealTime (dropdown)
 - StethoScope
 - Download to VxWorks target
- VxWorks**:
 - Base task priority: 30 (text input)
 - Task stack size: 16384 (text input)
- External mode options**:
 - External mode

In this section...

“Code Generation: Tornado Target Tab Overview” on page 4-280

“Standard math library” on page 4-281

“Code replacement library” on page 4-283

“Shared code placement” on page 4-285

“MAT-file logging” on page 4-287

“MAT-file variable name modifier” on page 4-289

“Code Format” on page 4-291

“StethoScope” on page 4-292

“Download to VxWorks target” on page 4-294

“Base task priority” on page 4-296

“Task stack size” on page 4-298

“External mode” on page 4-299

“Transport layer” on page 4-301

“MEX-file arguments” on page 4-303

“Static memory allocation” on page 4-305

“Static memory buffer size” on page 4-307

Code Generation: Tornado Target Tab Overview

Control Simulink Coder generated code for the Tornado® target.

Configuration

This tab appears only if you specify `tornado.tlc` as the “System target file” on page 4-6.

See Also

- *Tornado User’s Guide* from Wind River Systems
- *StethoScope User’s Guide* from Wind River Systems
- “Asynchronous Support”
- “Code Generation Pane: Tornado Target” on page 4-278

Standard math library

Specify a standard math library for your model.

Settings

Default: C89/C90 (ANSI)

C89/C90 (ANSI)

Generates calls to the ISO/IEC 9899:1990 C standard math library for floating-point functions.

C99 (ISO)

Generates calls to the ISO/IEC 9899:1999 C standard math library.

C++03 (ISO)

Generates calls to the ISO/IEC 14882:2003 C++ standard math library.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Dependencies

The C++03 (ISO) math library is available for use only if you select C++ for the **Language** parameter.

Command-Line Information

Parameter: TargetLangStandard

Type: string

Value: 'C89/C90 (ANSI)' | 'C99 (ISO)' | 'C++03 (ISO)'

Default: 'C89/C90 (ANSI)'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

“Specify Target Interfaces”

Code replacement library

Specify an application-specific math library for your model.

Settings

Default: None

None

Does not use a code replacement library.

GNU C99 extensions

Generates calls to the GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

Intel IPP

Generates calls to the Intel Performance Primitives (IPP) library.

Intel IPP/SSE with GNU99 extensions

Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions.

Note

- Additional values might be listed for licensed target products, for embedded and desktop targets, or if you have created and registered code replacement libraries using the Embedded Coder product.
- The list of **Code replacement library** values is filtered based on compatibility with the **Language**, **Standard math library**, and **Device vendor** values selected for your model.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Command-Line Information

Parameter: CodeReplacementLibrary

Type: string

Value: 'None' | 'GNU C99 extensions' | 'Intel IPP' | 'Intel IPP/SSE with GNU99 extensions'

Default: 'None'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

“Specify Target Interfaces”

Shared code placement

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Settings

Default: Auto

Auto

Operates as follows:

- When the model contains Model blocks, places utility code within the `slprj/target/_sharedutils` folder.
- When the model does not contain Model blocks, places utility code in the build folder (generally, in `model.c` or `model.cpp`).

Shared location

Directs code for utilities to be placed within the `slprj` folder in your working folder.

Command-Line Information

Parameter: UtilityFuncGeneration

Type: string

Value: 'Auto' | 'Shared location'

Default: 'Auto'

Recommended Settings

Application	Setting
Debugging	Shared location
Traceability	Shared location
Efficiency	No impact (execution, RAM) Shared location (ROM)
Safety precaution	No impact

See Also

- “Specify Target Interfaces”
- “Sharing Utility Code”

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: off



On

Enables MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in one of the following ways:

- Configuration Parameters dialog box, **Data Import/Export** pane, **Save to workspace** subpane (see “Data Import/Export Pane”)
- To Workspace blocks
- Scope blocks with the **Save data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in “Export Simulation Data” and “Configure Signal Data for Logging”. Setting MAT-file logging redirects the data to a MAT-file instead. The file is named *model.mat*, where *model* is the name of your model.



Off

Disables MAT-file logging. Clearing this option has the following benefits:

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

Dependencies

Selecting this parameter enables **MAT-file variable name modifier**.

Limitation

MAT-file logging does not work in a referenced model, and code is not generated to implement it.

Command-Line Information

Parameter: MatFileLogging

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

- “Logging”
- “Log Data for Analysis”
- “Virtualized Output Ports Optimization”

MAT-file variable name modifier

Select the string to add to the MAT-file variable names.

Settings

Default: `rt_`

`rt_` Adds a prefix string.

`_rt` Adds a suffix string.

`none` Does not add a string.

Dependency

If you have an Embedded Coder license, this parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: `LogVarNameModifier`

Type: `string`

Value: `'none' | 'rt_' | '_rt'`

Default: `'rt_'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Logging”
- “Log Data for Analysis”

Code Format

Specify the code generation format.

Settings

Default: RealTime

RealTime

Specifies the Real-Time code generation format.

RealTimeMalloc

Specifies the Real-Time Malloc code generation format.

Command-Line Information

Parameter: CodeFormat

Type: string

Value: 'RealTime' | 'RealTimeMalloc'

Default: 'RealTime'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Targets and Code Formats”

StethoScope

Specify whether to enable StethoScope, an optional data acquisition and data monitoring tool.

Settings

Default: off



On

Enables StethoScope.



Off

Disables StethoScope.

Tips

You can optionally monitor and change the parameters of the executing real-time program using either StethoScope or Simulink External mode, but not both with the same compiled image.

Dependencies

Enabling **StethoScope** automatically disables **External mode**, and vice versa.

Command-Line Information

Parameter: StethoScope

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact

Application	Setting
Efficiency	Off
Safety precaution	Off

See Also

- *Tornado User's Guide* from Wind River Systems
- *StethoScope User's Guide* from Wind River Systems

Download to VxWorks target

Specify whether to automatically download the generated program to the VxWorks target.

Settings

Default: off



On

Automatically downloads the generated program to VxWorks after each build.



Off

Does not automatically download to VxWorks, you must download generated programs manually.

Tips

- Automatic download requires specifying the target name and host name in the makefile.
- Before every build, reset VxWorks by pressing **Ctrl+X** on the host console or power-cycling the VxWorks chassis. This clears dangling processes or stale data that exists in VxWorks when the automatic download occurs.

Command-Line Information

Parameter: DownloadToVxWorks

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	Off

See Also

- *Tornado User's Guide* from Wind River Systems
- “Asynchronous Support”

Base task priority

Specify the priority with which the base rate task for the model is to be spawned.

Settings

Default: 30

Tips

- For a multirate, multitasking model, the Simulink Coder software increments the priority of each subrate task by one.
- The value you specify for this option will be overridden by a base priority specified in a call to the `rt_main()` function spawned as a task.

Command-Line Information

Parameter: BasePriority

Type: integer

Value: valid value

Default: 30

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Might impact efficiency, depending on other task's priorities
Safety precaution	No impact

See Also

- *Tornado User's Guide* from Wind River Systems

- “Asynchronous Support”

Task stack size

Stack size in bytes for each task that executes the model.

Settings

Default: 16384

Command-Line Information

Parameter: TaskStackSize

Type: integer

Value: valid value

Default: 16384

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Larger stack may waste space
Safety precaution	Larger stack reduces the possibility of overflow

See Also

- *Tornado User's Guide* from Wind River Systems
- "Asynchronous Support"

External mode

Specify whether to enable communication between the Simulink model and an application based on a client/server architecture.

Settings

Default: on



On

Enables External mode. The client (Simulink model) transmits messages requesting the server (application) to accept parameter changes or to upload signal data. The server responds by executing the request.



Off

Disables External mode.

Dependencies

Selecting this parameter enables:

- Transport layer
- MEX-file arguments
- Static memory allocation

Command-Line Information

Parameter: ExtMode

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

“Host/Target Communication”

Transport layer

Specify the transport protocol for External mode communications.

Settings

Default: tcpip

tcpip

Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. For targets provided by MathWorks, the value is specified in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`.

Dependency

This parameter is enabled by the **External mode** check box.

Command-Line Information

Parameter: ExtModeTransport

Type: integer

Value: 0

Default: 0

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Target Interfacing”

MEX-file arguments

Specify arguments to pass to an External mode interface MEX-file for communicating with executing targets.

Settings

Default: ''

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target (for example, 'myputer' or '148.27.151.12')
- Verbosity level (0 for no information or 1 for detailed information)
- TCP/IP server port number (an integer value between 256 and 65535, with a default of 17725)

Dependency

This parameter is enabled by the **External mode** check box.

Command-Line Information

Parameter: ExtModeMexArgs

Type: string

Value: valid arguments

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Target Interfacing”
- “Choose Communication Protocol for Client and Server”

Static memory allocation

Control the memory buffer for External mode communication.

Settings

Default: off



On

Enables the **Static memory buffer size** parameter for allocating allocate dynamic memory.



Off

Uses a static memory buffer for External mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- This parameter is enabled by the **External mode** check box.
- This parameter enables **Static memory buffer size**.

Command-Line Information

Parameter: ExtModeStaticAlloc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure External Mode Options for Code Generation”

Static memory buffer size

Specify the memory buffer size for External mode communication.

Settings

Default: 1000000

Enter the number of bytes to preallocate for External mode communications buffers in the target.

Tips

- If you enter too small a value for your application, External mode issues an out-of-memory error.
- To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information

Parameter: ExtModeStaticAllocSize

Type: integer

Value: valid value

Default: 1000000

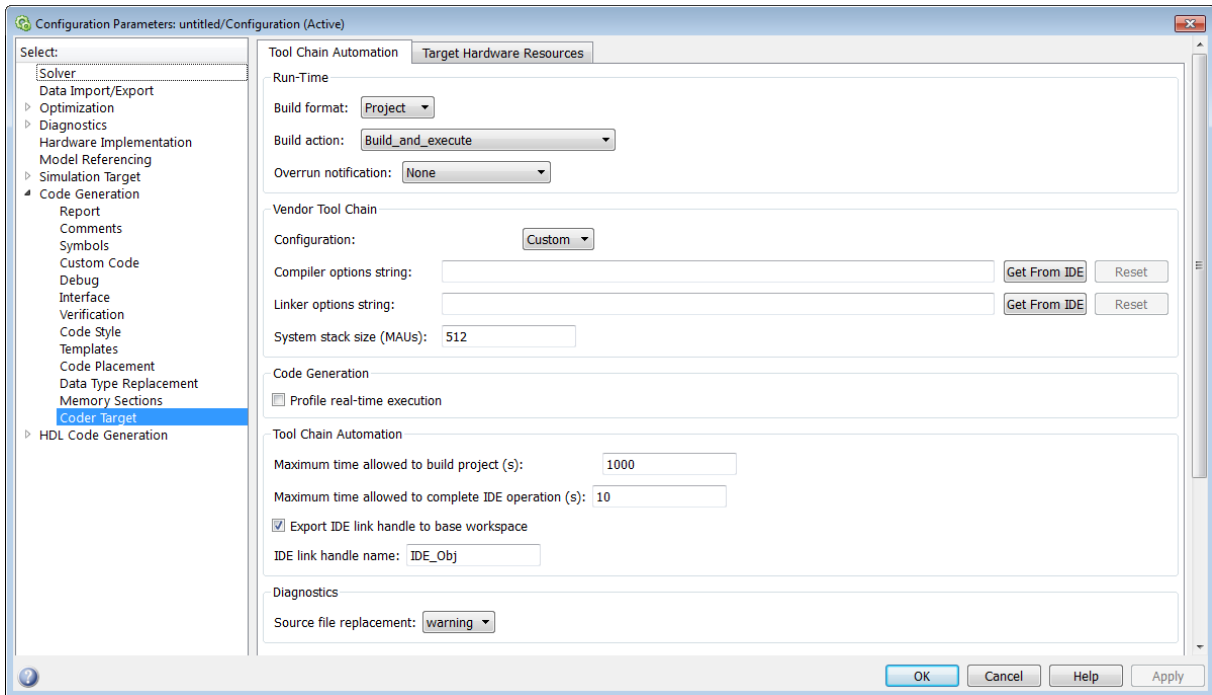
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Configure External Mode Options for Code Generation”

Code Generation: Coder Target Pane



In this section...

“Code Generation: Coder Target Pane Overview (previously “IDE Link Tab Overview”)” on page 4-311

“Coder Target: Tool Chain Automation Tab Overview” on page 4-312

“Build format” on page 4-314

“Build action” on page 4-316

“Overrun notification” on page 4-319

“Function name” on page 4-321

“Configuration” on page 4-322

“Compiler options string” on page 4-324

In this section...

“Linker options string” on page 4-326

“System stack size (MAUs)” on page 4-328

“Profile real-time execution” on page 4-331

“Profile by” on page 4-333

“Number of profiling samples to collect” on page 4-335

“Maximum time allowed to build project (s)” on page 4-337

“Maximum time allowed to complete IDE operation (s)” on page 4-339

“Export IDE link handle to base workspace” on page 4-340

“IDE link handle name” on page 4-342

“Source file replacement” on page 4-343

Code Generation: Coder Target Pane Overview (previously “IDE Link Tab Overview”)

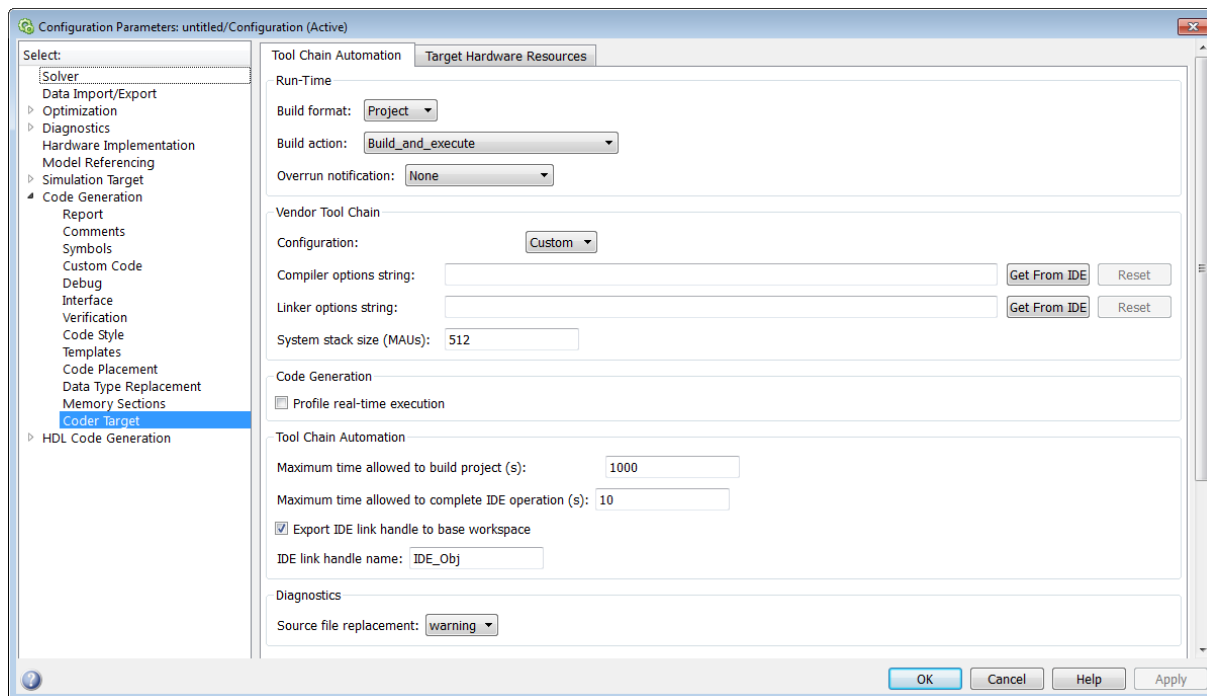
Configure the parameters for:

- Tool Chain Automation — How the coder software interacts with third-party software build toolchains.
- Target Hardware Resources — The IDE toolchain and properties of the physical hardware, such as board, operating system, memory, and peripherals.

See Also

- Coder Target: Tool Chain Automation Tab Overview
- Coder Target: Target Hardware Resources Tab Overview

Coder Target: Tool Chain Automation Tab Overview



The Tool Chain Automation Tab is only visible under the Coder Target pane.

The following table lists the parameters on the Tool Chain Automation Tab.

- “Build format” on page 4-314
- “Build action” on page 4-316
- “Overrun notification” on page 4-319
- “Function name” on page 4-321
- “Configuration” on page 4-322
- “Compiler options string” on page 4-324
- “Linker options string” on page 4-326
- “System stack size (MAUs)” on page 4-328

-
- “Profile real-time execution” on page 4-331
- “Profile by” on page 4-333
- “Number of profiling samples to collect” on page 4-335
- “Maximum time allowed to build project (s)” on page 4-337
- “Maximum time allowed to complete IDE operation (s)” on page 4-339
- “Export IDE link handle to base workspace” on page 4-340
- “IDE link handle name” on page 4-342
- “Source file replacement” on page 4-343

Build format

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Project

Project

Builds your model as an IDE project.

Makefile

Creates a makefile and uses it to build your model.

Dependencies

Selecting Makefile removes the following parameters:

- **Code Generation**
 - Profile real-time execution
 - Profile by
 - Number of profiling samples to collect
- **Link Automation**
 - Maximum time allowed to build project (s)
 - Maximum time allowed to complete IDE operation (s)
 - Export IDE link handle to base workspace
 - IDE link handle name

Command-Line Information

Parameter: buildFormat

Type: string

Value: Project | Makefile

Default: Build_and_execute

Recommended Settings

Application	Setting
Debugging	Project
Traceability	Project
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Build_and_execute

If you set **Build format** to Project, select one of the following options:

Build_and_execute

Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

Create_project

Directs Simulink Coder software to create a new project in the IDE. The command line equivalent for this setting is Create.

Archive_library

Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

Build

Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

Create_processor_in_the_loop_project

Directs the Simulink Coder code generation process to create PIL algorithm object code as part of the project build.

If you set **Build format** to Makefile, select one of the following options:

Create_makefile

Creates a makefile. For example, “.mk”. The command line equivalent for this setting is Create.

Archive_library

Creates a makefile and an archive library. For example, “.a” or “.lib”.

Build

Creates a makefile and an executable. For example, “.exe”.

Build_and_execute

Creates a makefile and an executable. Then it evaluates the execute instruction under the **Execute** tab in the current XMakefile configuration.

Dependencies

Selecting `Archive_library` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace**

Selecting `Create_processor_in_the_loop_project` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

Command-Line Information

Parameter: buildAction

Type: string

Value: Build | Build_and_execute | Create | Archive_library | Create_processor_in_the_loop_project

Default: Build_and_execute

Recommended Settings

Application	Setting
Debugging	Build_and_execute
Traceability	Archive_library
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

Overrun notification

Specifies how your program responds to overrun conditions during execution.

Settings

Default: None

None

Your program does not notify you when it encounters an overrun condition.

Print_message

Your program prints a message to standard output when it encounters an overrun condition.

Call_custom_function

When your program encounters an overrun condition, it executes a function that you specify in **Function name**.

Tips

- The definition of the standard output depends on your configuration.

Dependencies

Selecting Call_custom_function enables the **Function name** parameter.

Setting this parameter to Call_custom_function enables the **Function name** parameter.

Command-Line Information

Parameter: overrunNotificationMethod

Type: string

Value: None | Print_message | Call_custom_function

Default: None

Recommended Settings

Application	Setting
Debugging	Print_message or Call_custom_function
Traceability	Print_message
Efficiency	None
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Function name

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

Settings

No Default

Dependencies

This parameter is enabled by setting **Overrun notification** to `Call_custom_function`.

Command-Line Information

Parameter: `overrunNotificationFcn`

Type: string

Value: no default

Default: no default

Recommended Settings

Application	Setting
Debugging	String
Traceability	String
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Configuration

Sets the Configuration for building your project from the model.

Settings

Default: Custom

Custom

Lets the user apply a specialized combination of build and optimization settings.

Custom applies the same settings as the Release project configuration in IDE, except:

- The compiler options do not use optimizations.
- The memory configuration specifies a memory model that uses Far Aggregate for data and Far for functions.

Debug

Applies the Debug Configuration defined by the IDE to the generated project and code.

Release

Applies the Release project configuration defined by the IDE to the generated project and code.

Dependencies

- Selecting Custom disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting Release sets the **Compiler options string** to the settings defined by the IDE.
- Selecting Debug sets the **Compiler options string** to the settings defined by the IDE.

Command-Line Information

Parameter: projectOptions

Type: string

Value: Custom | Debug | Release

Default: Custom

Recommended Settings

Application	Setting
Debugging	Custom or Debug
Traceability	Custom, Debug, Release
Efficiency	Release
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the coder product does not set optimization flags.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Configuration** to **Custom** applies the **Custom** compiler options defined by coder software. **Custom** does not use optimizations.
- Setting **Configuration** to **Debug** applies the debug settings defined by the IDE.
- Setting **Configuration** to **Release** applies the release settings defined by the IDE.

Command-Line Information

Parameter: compilerOptionsStr

Type: string

Value: Custom | Debug | Release

Default: Custom

Recommended Settings

Application	Setting
Debugging	Custom
Traceability	Custom

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Linker options string

To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, the coder product does not set linker options.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information

Parameter: linkerOptionsStr

Type: string

Value: valid linker option

Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory.

This parameter is used in targets to allocate the stack size for the generated application. For example, with embedded processors that are not running an operating system, this parameter determines the total stack space that can be used for the application. For operating systems such as Linux or Windows, this value specifies the stack space allocated per thread.

This parameter also affects the “Maximum stack size (bytes)” parameter, located in the Optimization > Signals and Parameters pane.

Settings

Default: 8192

Minimum: 0

Maximum: Available memory

- Enter the stack size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.
- The software does not verify the value you entered is valid.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

When you set the **System target file** parameter on the **Code Generation** pane to `idmlink_ert.tlc` or `idmlink_grt.tlc`, the software sets the **Maximum stack size** parameter on the **Optimization > Signals and Parameters** pane to `Inherit from target` and makes it non-editable. In that case, the **Maximum stack size** parameter compares the value of **(System stack size/2)** with 200,000 bytes and uses the smaller of the two values.

Command-Line Information

Parameter: `systemStackSize`

Type: `int`

Default: 8192

Recommended Settings

Application	Setting
Debugging	<code>int</code>
Traceability	<code>int</code>
Efficiency	<code>int</code>
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Profile real-time execution

Enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

Settings

Default: Off



On

Adds instrumentation to the generated code to support execution profiling and generate the profiling report.



Off

Does not instrument the generated code to produce the profile report.

Dependencies

This parameter adds **Number of profiling samples to collect** and **Profile by**.

Selecting this parameter enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop` project removes this parameter.

Command-Line Information

Parameter: ProfileGenCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics..

Profile by

Defines which execution profiling technique to use.

Settings

Default: Task

Task

Profiles model execution by the tasks in the model.

Atomic subsystem

Profiles model execution by the atomic subsystems in the model.

Dependencies

Selecting **Real-time execution profiling** enables this parameter.

Command-Line Information

Parameter: profileBy

Type: string

Value: Task | Atomic subsystem

Default: Task

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics.

Number of profiling samples to collect

Specify the size of the buffer that holds the profiling samples. Enter a value that is 2 times the number of profiling samples.

Each task or subsystem execution instance represents one profiling sample. Each sample requires two memory locations, one for the start time and one for the end time. Consequently, the size of the buffer is twice the number of samples.

Sample collection begins with the start of code execution and ends when the buffer is full.

The profiling data is held in a statically sited buffer on the target processor.

Settings

Default: 100

Minimum: 2

Maximum: Buffer capacity

Tips

- Data collection stops when the buffer is full, but the application and processor continue running.
- Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

Dependencies

This parameter is enabled by **Profile real-time execution**.

Command-Line Information

Parameter: ProfileNumSamples

Type: int

Value: Positive integer

Default: 100

Recommended Settings

Application	Setting
Debugging	100
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

Settings

Default: 1000

Minimum: 1

Maximum: No limit

Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to complete IDE operation** timeout value.

Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

Command-Line Information

Parameter: `ideObjBuildTimeout`

Type: int

Value: Integer greater than 0

Default: 100

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Maximum time allowed to complete IDE operation (s)

specifies how long, in seconds, the software waits for IDE functions, such as read or write, to return completion messages.

Settings

Default: 10

Minimum: 1

Maximum: No limit

Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to build project (s)** timeout value

Command-Line Information

Parameter: 'ideObjTimeout'

Type: int

Value:

Default: 10

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Export IDE link handle to base workspace

Directs the software to export the IDE_Obj object to your MATLAB workspace.

Settings

Default: On



On

Directs the build process to export the IDE_Obj object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.



Off

prevents the build process from exporting the IDE_Obj object to your MATLAB software workspace.

Dependency

Selecting **Profile real-time execution** enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Selecting **Export IDE link handle to base workspace** enables **IDE link handle name**.

Command-Line Information

Parameter: exportIDEObj

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

IDE link handle name

specifies the name of the IDE_Obj object that the build process creates.

Settings

Default: IDE_Obj

- Enter a valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the IDE_Obj object.
- The handle name is case sensitive.

Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

Command-Line Information

Parameter: ideObjName

Type: string

Value:

Default: IDE_Obj

Recommended Settings

Application	Setting
Debugging	Enter a valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Source file replacement

Selects the diagnostic action to take if the coder software detects conflicts that you are replacing source code with custom code.

Settings

Default: warn

none

Does not generate warnings or errors when it finds conflicts.

warning

Displays a warning.

error

Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

Tips

- The build operation continues if you select `warning` and the software detects custom code replacement. You see warning messages as the build progresses.
- Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select `none` when you do not want to see multiple messages during your build.
- The messages apply to Simulink Coder **Custom Code** replacement options as well.

Command-Line Information

Parameter: DiagnosticActions

Type: string

Value: none | warning | error

Default: warning

Recommended Settings

Application	Setting
Debugging	error
Traceability	error
Efficiency	warning
Safety precaution	error

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Parameter Reference

In this section...

“Recommended Settings Summary” on page 4-345

“Parameter Command-Line Information Summary” on page 4-375

Recommended Settings Summary

The following table summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the GRT and ERT targets, unless otherwise specified.

For parameters that are available only when an ERT target is specified, see the “Recommended Settings Summary” in the Embedded Coder documentation.

For additional details, click the links in the Configuration Parameter column.

Mapping Application Requirements to the Solver Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Start time	No impact	No impact	No impact	0.0	0.0 seconds
Stop time	No impact	No impact	No impact	A positive value	10.0 seconds
Type	Fixed-step	Fixed-step	Fixed-step	Fixed-step	Variable-step (you must change to Fixed-step for code generation)

Mapping Application Requirements to the Solver Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Solver”	No impact	No impact	No impact	Discrete (no continuous states)	ode3 (Bogacki-Shampine)
“Periodic sample time constraint”	No impact	No impact	No impact	Specified or Ensure sample time independent	Unconstrained
“Sample time properties”	No impact	No impact	No impact	Period, offset, and priority of each sample time in the model; faster sample times must have higher priority than slower sample times	''

Mapping Application Requirements to the Solver Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Tasking mode for periodic sample times	No impact	No impact	No impact	No impact	Auto
“Automatically handle rate transition for data transfer”	No impact	No impact (for simulation and during development) Off (for production code generation)	No impact	Off	Off

Mapping Application Requirements to the Data Import/Export Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Input”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
“Initial state”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off

Mapping Application Requirements to the Data Import/Export Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Time”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On
“States”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
“Output”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On
“Final states”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
“Signal logging”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On
“Record and inspect simulation output”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
“Limit data points to last”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On

Mapping Application Requirements to the Data Import/Export Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Decimation”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	1
“Format”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Array
“Output options”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Refine output
“Refine factor”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	1
“Output times”	No impact	No impact	No impact	No impact (GRT) Off (ERT)	' [] '

Mapping Application Requirements to the Optimization Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Block reduction	Off (GRT) No impact (ERT)	Off	On	Off	On
Implement logic signals as Boolean data (vs. double)	No impact	No impact	On	On	On
Conditional input branch execution	No impact	On	On (execution) No impact (ROM, RAM)	No impact	On
Application lifespan (days)	No impact	No impact	Finite value	inf	inf
Use memset to initialize floats and doubles to 0.0	No impact	No impact	On* (execution, ROM) No impact (RAM)	No impact	On
Use floating-point multiplication to handle net slope corrections	No impact	No impact	On (when target hardware supports efficient multiplication) Off (otherwise)	Off	Off

Mapping Application Requirements to the Optimization Pane: General Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Remove code from floating-point to integer conversions that wraps out-of-range values	Off	Off	On (execution, ROM) No impact (RAM)	Off (GRT) On (ERT)	Off
Remove code from floating-point to integer conversions with saturation that maps NaN to zero	Off	Off	On	Off (GRT) On (ERT)	On

*The command-line value is reverse of the listed value.

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Inline parameters	Off (GRT) On (ERT)	On	On	No impact	Off
Signal storage reuse	Off	Off	On	No impact	On

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Enable local block outputs	Off	No impact	On	No impact	On
Eliminate superfluous local variables (Expression folding)	Off	No impact (GRT) Off (ERT)	On	No impact	On
“Optimize global data access”	Off	Off	No impact (execution) On (ROM, RAM)	No impact	Off
Loop unrolling threshold	No impact	No impact	>0	>1	5
Maximum stack size (bytes)	No impact	No impact	No impact	No impact	Inherit from target
Use memcpy for vector assignment	No impact	No impact	On	No impact	On
Memcpy threshold (bytes)	No impact	No impact	Accept default or determine target-specific optimal value	No impact	64

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Reuse local block outputs	Off	Off	On	No impact	On
Inline invariant signals	Off	Off	On	No impact	Off

Mapping Application Requirements to the Optimization Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Use bitsets for storing state configuration”	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off
“Use bitsets for storing Boolean data”	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off

Mapping Application Requirements to the Diagnostics Pane: Solver Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Algebraic loop”	error	No impact	No impact	error	warning
“Minimize algebraic loop”	No impact	No impact	No impact	error	warning
“Block priority violation”	No impact	No impact	No impact	error	warning
“Consecutive zero-crossings violation”	No impact	No impact	No impact	warning or error	error
“Unspecified inheritability of sample time”	No impact	No impact	No impact	error	warning
“Solver data inconsistency”	warning	No impact	none	No impact	warning
“Automatic solver parameter selection”	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Sample Time Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Source block specifies -1 sample time”	No impact	No impact	No impact	error	none
“Discrete used as continuous”	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Sample Time Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Multitask rate transition”	No impact	No impact	No impact	error	error
“Single task rate transition”	No impact	No impact	No impact	none or error	none
“Multitask conditionally executed subsystem”	No impact	No impact	No impact	error	error
“Tasks with equal priority”	No impact	No impact	No impact	none or error	warning
“Enforce sample times specified by Signal Specification blocks”	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Signal resolution”	No impact	No impact	No impact	Explicit only	Explicit only
“Division by singular matrix”	No impact	No impact	No impact	error	none
“Underspecified data types”	No impact	No impact	No impact	error	none

**Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab
(Continued)**

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Simulation range checking”	warning or error	warning or error	none	error	none
“Detect overflow”	No impact	No impact	No impact	error	warning
“Inf or NaN block output”	No impact	No impact	No impact	error	none
““rt” prefix for identifiers”	No impact	No impact	No impact	error	error
“Detect downcast”	No impact	No impact	No impact	error	error
“Detect overflow”	No impact	No impact	No impact	error	error
“Detect underflow”	No impact	No impact	No impact	error	none
“Detect precision loss”	No impact	No impact	No impact	error	error
“Detect loss of tunability”	No impact	No impact	No impact	error	none
“Detect read before write”	No impact	No impact	No impact	error	Enable all as warnings
“Detect write after read”	No impact	No impact	No impact	error	Enable all as warning
“Detect write after write”	No impact	No impact	No impact	error	Enable all as errors
“Multitask data store”	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Duplicate data store names”	warning	No impact	none	No impact	none
“Check undefined subsystem initial output”	No impact	No impact	No impact	On	On
“Check preactivation output of execution context”	No impact	No impact	No impact	On	Off
“Check runtime output of execution context”	No impact	No impact	No impact	On	Off
Model Verification block enabling	No impact	No impact	No impact	No impact (GRT) Disable all (ERT)	Use local settings

Mapping Application Requirements to the Diagnostics Pane: Type Conversion Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Unnecessary type conversions”	No impact	No impact	No impact	warning	none
“Vector/matrix block input conversion”	No impact	No impact	No impact	error	none
“32-bit integer to single precision float conversion”	No impact	No impact	No impact	warning	warning

Mapping Application Requirements to the Diagnostics Pane: Connectivity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Signal label mismatch”	No impact	No impact	No impact	error	none
“Unconnected block input ports”	No impact	No impact	No impact	error	warning
“Unconnected block output ports”	No impact	No impact	No impact	error	warning
“Unconnected line”	No impact	No impact	No impact	error	none
“Unspecified bus object at root Outport block”	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Connectivity Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Element name mismatch”	No impact	No impact	No impact	error	warning
“Mux blocks used to create bus signals”	No impact	No impact	No impact	error	error
“Bus signal treated as vector”	No impact	No impact	No impact	error	warning
“Invalid function-call connection”	No impact	No impact	No impact	error	error
“Context-dependent inputs”	No impact	No impact	No impact	Enable all	Use local settings

Mapping Application Requirements to the Diagnostics Pane: Compatibility Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“S-function upgrades needed”	No impact	No impact	No impact	error	none

Mapping Application Requirements to the Diagnostics Pane: Model Referencing Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Model block version mismatch”	No impact	No impact	No impact	none	none
“Port and parameter mismatch”	No impact	No impact	No impact	error	none
“Model configuration mismatch”	No impact	No impact	No impact	warning	none
“Invalid root Inport/Outport block connection”	No impact	No impact	No impact	error	none
“Unsupported data logging”	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Saving Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Block diagram contains disabled library links”	No impact	No impact	No impact	No impact	warning
“Block diagram contains parameterized library links”	No impact	No impact	No impact	No impact	none

Mapping Application Requirements to the Diagnostics Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Unused data and events”	warning	No impact	No impact (for simulation and during development) none (for production code generation)	warning	warning
“Unexpected backtracking”	warning	No impact	No impact	error	warning
“Invalid input data access in chart initialization”	warning	No impact	No impact	error	warning
“No unconditional default transitions”	warning	No impact	No impact (for simulation and during development) none (for production	error	warning

Mapping Application Requirements to the Diagnostics Pane: Stateflow Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
			code generation)		
“Transition outside natural parent”	warning	No impact	No impact (for simulation and during development) none (for production code generation)	error	warning

Mapping Application Requirements to the Hardware Implementation Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Device vendor	No impact	No impact	No impact	No impact	Generic
Device type	No impact	No impact	No impact	No impact	Unspecified (assume 32 bit Generic)

Mapping Application Requirements to the Hardware Implementation Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Number of bits	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for code generation	char 8, short 16, int 32, long 32, long long 64, float 32, double 64, native 32, pointer 32
Largest atomic size	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for code generation	integer Char, floating-point None
Byte ordering	No impact	No impact	No impact	No impact	Unspecified

Mapping Application Requirements to the Hardware Implementation Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Signed integer division rounds to	No impact for simulation and during development Undefined for production code generation	No impact for simulation and during development Zero or Floor for production code generation	No impact for simulation and during development Zero for production code generation	No impact for simulation and during development Floor for production code generation	Undefined
Shift right on a signed integer as arithmetic shift	No impact	No impact	On	No impact	On
Enable long long	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for production code generation	Off
Test hardware is the same as production hardware	No impact	No impact	No impact	No impact	On

Mapping Application Requirements to the Model Referencing Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Rebuild”	No impact	No impact	No impact	If any changes detected or Never If you use the Never setting, then set the Never rebuild diagnostic parameter to Error if rebuild required	If any changes detected
“Never rebuild diagnostic”	No impact	No impact	No impact	error if rebuild required	error if rebuild required
“Enable parallel model reference builds”	No impact	No impact	No impact	No impact	Off
“MATLAB worker initialization for builds”	No impact	No impact	No impact	No impact	None
“Total number of instances allowed per top model”	No impact	No impact	No impact	No impact	Multiple

Mapping Application Requirements to the Model Referencing Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Pass fixed-size scalar root inputs by value for code generation”	No impact	No impact	No impact	Off	Off
“Minimize algebraic loop occurrences”	No impact	No impact	No impact	Off	Off
“Propagate sizes of variable-size signals”	No impact	No impact	No impact	Off	Infer from blocks in model
“Model dependencies”	No impact	No impact	No impact	No impact	' '

Mapping Application Requirements to the Simulation Target Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Enable debugging/animation”	On	No impact	Off	On	On
“Enable overflow detection (with debugging)”	On	No impact	Off	On	On
“Ensure memory integrity”	On	On	Off	On	On

Mapping Application Requirements to the Simulation Target Pane: General Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Echo expressions without semicolons”	On	No impact	Off	No impact	On
“Ensure responsiveness”	On	On	Off	On	On
“Simulation target build mode”	No impact	No impact	No impact	No impact	Incremental build

Mapping Application Requirements to the Simulation Target Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Reserved names”	No impact	No impact	No impact	No impact	{}

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Parse custom code symbols”	On	No impact	No impact	On	On
“Source file”	No impact	No impact	No impact	No impact	' '

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
“Header file”	No impact	No impact	No impact	No impact	''
“Initialize function”	No impact	No impact	No impact	No impact	''
“Terminate function”	No impact	No impact	No impact	No impact	''
“Include directories”	No impact	No impact	No impact	No impact	''
“Source files”	No impact	No impact	No impact	No impact	''
“Libraries”	No impact	No impact	No impact	No impact	''

Mapping Application Requirements to the Code Generation Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
System target file	No impact	No impact	No impact	No impact (GRT) ERT based (ERT)	grt.tlc
Language	No impact	No impact	No impact	No impact	C

Mapping Application Requirements to the Code Generation Pane: General Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Compiler optimization level	Optimizations off (faster builds)	Optimizations off (faster builds)	Optimizations on (faster runs) (execution) No impact (ROM, RAM)	No impact	Optimizations off (faster builds)
Custom compiler optimization flags	Optimizations off (faster builds)	Optimizations off (faster builds)	Optimizations on (faster runs)	No impact	Optimizations off (faster builds)
Generate makefile	No impact	No impact	No impact	No impact	On
Make command	No impact	No impact	No impact	make_rtw	make_rtw
Template makefile	No impact	No impact	No impact	No impact	grt_default_tmf
“Select objective” on page 4-32	Debugging	Not applicable for GRT-based targets	Execution efficiency	Not applicable for GRT-based targets	Unspecified
“Check model before generating code” on page 4-40	On (proceed with warnings) or On (stop for warnings)	On (proceed with warnings) or On (stop for warnings)	On (proceed with warnings) or On (stop for warnings)	On (proceed with warnings) or On (stop for warnings)	Off
Generate code only	Off	No impact	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precautions	
“Create code generation report” on page 4-52	On	On	No impact	On	Off
“Open report automatically” on page 4-55	On	On	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Comments Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Include comments	On	On	No impact	On	On
Simulink block / Stateflow object comments	On	On	No impact	On	On
Show eliminated blocks	On	On	No impact	On	Off
Verbose comments for Simulink Global storage class	On	On	No impact	On	Off
Operator Annotations	No impact	On	No impact	On	Off

Mapping Application Requirements to the Code Generation Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Maximum identifier length	Valid value	>30	No impact	>30	31
Use the same reserved names as Simulation Target	No impact	No impact	No impact	No impact	Off
Reserved names	No impact	No impact	No impact	No impact	{ }

Mapping Application Requirements to the Code Generation Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Use the same custom code settings as Simulation Target	No impact	No impact	No impact	No impact	Off
Source file	No impact	No impact	No impact	No impact	''
Header file	No impact	No impact	No impact	No impact	''
Initialize function	No impact	No impact	No impact	No impact	''
Terminate function	No impact	No impact	No impact	No impact	''
Include directories	No impact	No impact	No impact	No impact	''

Mapping Application Requirements to the Code Generation Pane: Custom Code Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Source files	No impact	No impact	No impact	No impact	''
Libraries	No impact	No impact	No impact	No impact	''

Mapping Application Requirements to the Code Generation Pane: Debug Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Verbose build	On	No impact	No impact	On	On
Retain .rtw file	On	No impact	No impact	No impact	Off
“Profile TLC” on page 4-172	On	No impact	No impact	No impact	Off
Start TLC debugger when generating code	On	No impact	No impact	No impact	Off
Start TLC coverage when generating code	On	No impact	No impact	No impact	Off
Enable TLC assertion	On	No impact	No impact	On	Off

Mapping Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Standard math library	No impact	No impact	Valid library	No impact	C89/C90 (ANSI)
Code replacement library	No impact	No impact	Valid library	No impact	None
Shared code placement	Shared location (GRT) No impact (ERT)	Shared location (GRT) No impact (ERT)	No impact (execution, RAM) Shared location (ROM)	No impact	Auto
Support non-finite numbers	No impact	No impact	Off (Execution, ROM) No impact (RAM)	Off	On
Code interface packaging	No impact	No impact	Reusable function or C++ class	No impact	Nonreusable function if Language is set to C; C++ class if Language is set to C++
Multi-instance code error diagnostic	Warning or Error	No impact	None	No impact	Error

Mapping Application Requirements to the Code Generation Pane: Interface Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Classic call interface	No impact	Off	Off (execution, ROM), No impact (RAM)	Off	Off (except On for GRT models created before R2012a)
MAT-file logging	On	No impact	Off	Off	On (GRT) Off (ERT)
MAT-file variable name modifier	No impact	No impact	No impact	No impact	rt_
Interface	No impact	No impact	No impact	No impact (GRT) None (ERT)	None
Generate C API for: signals	No impact	No impact	No impact	No impact	On
Generate C API for: parameters	No impact	No impact	No impact	No impact	On
Generate C API for: states	No impact	No impact	No impact	No impact	Off
Generate C API for: root-level I/O	No impact	No impact	No impact	No impact	Off
Transport layer	No impact	No impact	No impact	No impact	tcpip
MEX-file arguments	No impact	No impact	No impact	No impact	' '

Mapping Application Requirements to the Code Generation Pane: Interface Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Static memory allocation	No impact	No impact	No impact	No impact	Off
“Static memory buffer size” on page 4-307	No impact	No impact	No impact	No impact	1000000

Parameter Command-Line Information Summary

The following table lists Simulink Coder parameters that you can use to tune model and target configurations. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to Configuration Parameter dialog box equivalents.

Use the `get_param` and `set_param` commands to retrieve and set the values of the parameters on the MATLAB command line or programmatically in scripts.

The Configuration Wizard in the Embedded Coder product provides buttons and scripts for customizing code generation. For information on using Configuration Wizard features, see “Use Configuration Wizard Blocks” in the Embedded Coder documentation.

For general information about Simulink parameters, see “Configuration Parameters Dialog Box Overview”. For information on using `get_param` and `set_param` to tune the parameters for various model configurations, see “Tune Parameters”.

For parameters that are specific to the ERT target, or targets based on the ERT target, see “Parameter Command-Line Information Summary” in the Embedded Coder documentation.

Note Parameters that are specific to Stateflow or Fixed-Point Designer™ products are marked with (Stateflow) and (Fixed-Point Designer), respectively.

The default setting for a parameter might vary for different targets.

Command-Line Information: Optimization Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
BooleanDataType off, on	Optimization > Implement logic signals as Boolean data (vs. double)	Control the output data type of blocks that generate logic signals.
EfficientFloat2IntCast off , on	Optimization > Remove code from floating-point to integer conversions that wrap out-of-range values	Remove wrapping code that handles out-of-range floating-point to integer conversion results.
EfficientMapNaN2IntZero off, on	Optimization > 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.
InitFltsAndDblsToZero off , on	Optimization > Use memset to initialize floats and doubles to 0.0	Optimize initialization of storage for float and double values. Set this option if the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern 0.
LifeSpan <i>string</i>	Optimization > Application lifespan (days)	Optimize the size of counters used to compute absolute and elapsed time, using the specified application life span value.
NoFixptDivByZeroProtection (Fixed-Point Designer) off , on	Optimization > Remove code that protects against division arithmetic exceptions	Suppress generation of code that guards against division by zero for fixed-point data.

Command-Line Information: Optimization Pane: General Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
UseFloatMulNetSlope (Fixed-Point Designer) off , on	Optimization > Use floating-point multiplication to handle net slope corrections	Use floating-point multiplication to perform net slope correction for floating-point to fixed-point casts.
UseIntDivNetSlope (Fixed-Point Designer) off , on	Optimization > Use integer division to handle net slopes that are reciprocals of integers	Perform net slope correction using integer division when simplicity and accuracy conditions are met.

Command-Line Information: Optimization Pane: Signals and Parameters Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
BufferReuse off, on	Optimization > Signals and Parameters > Reuse local block outputs	Reuse local (function) variables for block outputs wherever possible. Selecting this option trades code traceability for code efficiency.
EnableMemcpy off, on	Optimization > Signals and Parameters > Use memcpy for vector assignment	Optimize code generated for vector assignment by replacing for loops with memcpy function calls.

Command-Line Information: Optimization Pane: Signals and Parameters Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
EnhancedBackFolding off, on	Optimization > Signals and Parameters > Minimize data copies between local and global variables	Reuse existing global variables to store temporary results.
ExpressionFolding off, on	Optimization > Signals and Parameters > Eliminate superfluous local variables (Expression folding) > Interface	Collapse block computations into single expressions wherever possible. This improves code readability and efficiency.
InlineInvariantSignals off, on	Optimization > Signals and Parameters > Inline invariant signals	Precompute and inline the values of invariant signals in the generated code.
LocalBlockOutputs off, on	Optimization > Signals and Parameters > Enable local block outputs	Declare block outputs in local (function) scope wherever possible to reduce global RAM usage.
MemcpyThreshold int - 64	Optimization > Signals and Parameters > Memcpy threshold (bytes)	Specify the minimum array size in bytes for which memcpy function calls should replace for loops in the generated code for vector assignments.

Command-Line Information: Optimization Pane: Signals and Parameters Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
RollThreshold int - 5	Optimization > Signals and Parameters > Loop unrolling threshold	Specify the minimum signal width for which a for loop is to be generated.
MaxStackSize <Specify a value>, Inherit from target	Optimization > Signals and Parameters > Maximum stack size (bytes)	Specify the maximum stack size in bytes for your model.

Command-Line Information: Optimization Pane: Stateflow Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
DataBitsets (Stateflow) off , on	Optimization > Stateflow > Use bitsets for storing Boolean data	Use bit sets for storing Boolean data.
StateBitsets (Stateflow) off , on	Optimization > Stateflow > Use bitsets for storing state configuration	Use bit sets for storing state configuration.

Command-Line Information: Code Generation Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CheckMdlBeforeBuild <i>string</i> - off , warning, error	Code Generation > Check model before generating code	Specify whether to run Code Generation Advisor checks before generating code.
GenCodeOnly <i>string</i> - off , on	Code Generation > Generate code only	Generate source code, but do not execute the makefile to build an executable.
GenerateMakefile <i>string</i> - off, on	Code Generation > Generate makefile	Specify whether to generate a makefile during the build process for a model.
MakeCommand <i>string</i> - make_rtw	Code Generation > Make command	Specify the make command and optional arguments to be used to generate an executable for the model.
ObjectivePriorities (GRT) <i>string</i> - {''}, {'Debugging'}, {'Execution efficiency'}	Code Generation > Select objective	Specify the code generation objectives to use with the Code Generation Advisor.
ObjectivePriorities (ERT) <i>string</i> - {''}, {'Efficiency'}, {'Traceability'}, {'Safety precaution'}, {'Debugging'}	Code Generation > Set Objectives	Specify and prioritize the code generation objectives to use with the Code Generation Advisor.

Command-Line Information: Code Generation Pane: General Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
RTWCompilerOptimization string - Off , On, Custom	Code Generation > Compiler optimization level	Use this parameter to trade off compilation time against run time for your model code without having to supply compiler-specific flags to other levels of the Simulink Coder build process. Off - Turn compiler optimizations off for faster builds On - Turn compiler optimizations on for faster code execution Custom - Specify custom compiler optimization flags via the RTWCustomCompilerOptimizations parameter
RTWCustomCompilerOptimizations string - '', unquoted string of compiler optimization flags	Code Generation > Custom compiler optimization flags	If you specified Custom to the RTWCompilerOptimization parameter, use this parameter to specify custom compiler optimization flags, for example, -O2.
SaveLog off , on	Code Generation > Save build log	Save build log.
SystemTargetFile string - grt.tlc	Code Generation > System target file	Specify a system target file.
TargetLang string - C , C++	Code Generation > Language	Specify whether to generate C or C++ code.
TemplateMakefile string - grt_default_tmf	Code Generation > Template makefile	Specify the current template makefile for building a Simulink Coder target.

Command-Line Information: Code Generation Pane: Report Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
GenerateReport string - off , on	Code Generation > Report > Create code generation report	Document the generated C or C++ code in an HTML report.
LaunchReport string - off , on	Code Generation > Report > Launch report automatically	Display the HTML report after code generation completes.

Command-Line Information: Code Generation Pane: Comments Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ForceParamTrailComments string - off , on	Code Generation > Comments > Verbose comments for SimulinkGlobal storage class	Specify that comments be included in the generated file. To reduce file size, the model parameters data structure is not commented when there are more than 1000 parameters.
GenerateComments string - off , on	Code Generation > Comments > Include comments	Include comments in generated code.
OperatorAnnotations string - off , on	Code Generation > Comments > Operator annotations	Specify whether to include operator annotations in the generated code as comments.

Command-Line Information: Code Generation Pane: Comments Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ShowEliminatedStatement <i>string</i> - off , on	Code Generation > Comments > Show eliminated blocks	Show statements for eliminated blocks as comments in the generated code.
SimulinkBlockComments <i>string</i> - off , on	Code Generation > Comments > Simulink block / Stateflow object comments	Insert Simulink block and Stateflow object names as comments above the generated code for each block.

Command-Line Information: Code Generation Pane: Symbols Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
MaxIdLength int - 31	Code Generation > Symbols > Maximum identifier length	Specify the maximum number of characters that can be used in generated function, type definition, and variable names.
ReservedNameArray <i>string array</i> - {}	Code Generation > Symbols > 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 to avoid name conflicts.
UseSimReservedNames string - off , on	Code Generation > Symbols > Use the same reserved names as Simulation Target	Specify whether to use the same reserved names as those specified in the Simulation Target > Symbols pane.

Command-Line Information: Code Generation Pane: Custom Code Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CustomHeaderCode <i>string</i> - ''	Code Generation > Custom Code > Header file	Specify code to appear near the top of the generated model header file.
CustomInclude <i>string</i> - ''	Code Generation > Custom Code > Include directories	Specify a space-separated list of include folders to add to the include path when compiling the generated code. Note If your list includes Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, 'C:\Project "C:\Custom Files"'
CustomInitializer <i>string</i> - ''	Code Generation > Custom Code	Specify code to appear in the generated model initialize function.
CustomLibrary <i>string</i> - ''	Code Generation > Custom Code > Initialize function Libraries	Specify a space-separated list of static library files to link with the generated code.
CustomSource <i>string</i> - ''	Code Generation > Custom Code > Source files	Specify a space-separated list of source files to compile and link with the generated code.
CustomSourceCode <i>string</i> - ''	Code Generation > Custom Code > Source file	Specify code to appear near the top of the generated model source file.

Command-Line Information: Code Generation Pane: Custom Code Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CustomTerminator string - ''	Code Generation > Custom Code > Terminate function	Specify code to appear in the generated model terminate function.
RTWUseSimCustomCode string - off , on	Code Generation > Custom Code > 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.

Command-Line Information: Code Generation Pane: Debug Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ProfileTLC string - off , on	Code Generation > Debug > Profile TLC	Profile the execution time of each TLC file used to generate code for this model in HTML format.
RTWVerbose string - off , on	Code Generation > Debug > Verbose build	Display messages indicating code generation stages and compiler output.
RetainRTWFile string - off , on	Code Generation > Debug > Retain .rtw file	Retain the <i>model</i> .rtw file in the current build folder.
TLCAssert string - off , on	Code Generation > Debug > Enable TLC assertion	Produce a TLC stack trace when the argument to the <code>assert</code> directives evaluates to false.

Command-Line Information: Code Generation Pane: Debug Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
TLCCoverage string - off , on	Code Generation > Debug > Start TLC coverage when generating code	Generate .log files containing the number of times each line of TLC code is executed during code generation.
TLCDebug string - off , on	Code Generation > Debug > Start TLC debugger when generating code	Start the TLC debugger during code generation at the beginning of the TLC program. TLC breakpoint statements automatically invoke the TLC debugger regardless of this setting.

Command-Line Information: Code Generation Pane: Interface Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CodeInterfacePackaging string - C++ class, Nonreusable function, Resuable function (Default is Nonreusable function if TargetLang is set to C, or C++ class if TargetLang is set to C++)	Code Generation > Interface > Code interface packaging	Specify the packaging for the generated C or C++ code interface.
CodeReplacementLibrary string - None , GNU C99 extensions, Intel IPP, Intel IPP/SSE with GNU99 extensions	Code Generation > Interface > Code replacement library	Specify an application-specific math library for your model. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur. None

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
(For ERT-based models, additional values may be available; see the Code replacement library drop-down list in the Configuration Parameters dialog box.)		GNU C99 extensions - GNU gcc math library, which provides C99 extensions as defined by compiler option <code>-std=gnu99</code> Intel IPP - Intel Performance Primitives (IPP) library Intel IPP/SSE with GNU99 extensions - GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions
ExtMode off , on	Code Generation > Interface > Interface	Specify the data interface to be generated with the code.
ExtModeMexArgs <i>string</i> ('')	Code Generation > Interface > Interface > External mode > MEX-file arguments	Specify arguments that are passed to an external mode interface MEX-file for communicating with executing targets.
ExtModeStaticAlloc off , on	Code Generation > Interface > Static memory allocation	Use a static memory buffer for external mode instead of allocating dynamic memory (calls to <code>malloc</code>).
ExtModeStaticAllocSize <i>integer</i> (1000000)	Code Generation > Interface > Static memory buffer size	Specify the size in bytes of the external mode static memory buffer.

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ExtModeTransport int - 0 for TCP/IP, 1 for serial	Code Generation > Interface > Interface > External mode > Transport layer	Specify transport protocols for external mode communications.
GenerateASAP2 off , on	Code Generation > Interface > Interface	Specify the data interface to be generated with the code.
GRTInterface string - off (except on for GRT models created before R2012a), on	Code Generation > Interface > Classic call interface	Include a code interface (wrapper) that is compatible with the pre-R2012a GRT target.
LogVarNameModifier string - none , rt_, _rt	Code Generation > Interface > MAT-file variable name modifier	Augment the MAT-file variable name.
MatFileLogging string - off, on (Default is on for GRT targets, off for ERT targets)	Code Generation > Interface > MAT-file logging	Generate code that logs data to a MAT-file.
MultiInstanceErrorCode string - None, Warning, Error	Code Generation > Interface Multi-instance code error diagnostic	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of multi-instance code.
RTWCAPIParams string - off , on	Code Generation > Interface > Generate C API for: parameters	Generate C API parameter tuning structures.

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
RTWCAPIRootIo string - off , on	Code Generation > Interface > Generate C API for: root-level I/O	Generate a C API root-level I/O structure
RTWCAPISignals string - off , on	Code Generation > Interface > Generate C API for: signals	Generate C API signal structure.
RTWCAPIStates string - off , on	Code Generation > Interface > Generate C API for: states	Generate C API state structure.
SupportNonFinite string - off, on	Code Generation > Interface > Support non-finite numbers	Support nonfinite values (inf, nan, -inf) in the generated code.
TargetLangStandard string - C89/C90 (ANSI) , C99 (ISO), C++03 (ISO)	Code Generation > Interface > Standard math library	Specify a standard math library for your model. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur. C89/C90 (ANSI) - ISO/IEC 9899:1990 C standard math library for floating-point functions C99 (ISO) - ISO/IEC 9899:1999 C standard math library

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		C++03 (ISO) - ISO/IEC 14882:2003 C++ standard math library
UtilityFuncGeneration string - Auto , Shared location	Code Generation > Interface > Shared code placement	Specify where utility code is to be generated.

Command-Line Information: Not in GUI

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CodeGenDirectory	Not available	For MathWorks use only.
Comment	Not available	For MathWorks use only.
CompOptLevelCompliant off, on	Not available	Set in SelectCallback for a target to indicate whether the target supports the ability to use the Compiler optimization level parameter on the Code Generation pane 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.
ConfigAtBuild	Not available	For MathWorks use only.
ConfigurationMode	Not available	For MathWorks use only.
ConfigurationScript	Not available	For MathWorks use only.

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ERTCustomFileBanners	Not available	For MathWorks use only.
EvaledLifeSpan	Not available	For MathWorks use only.
ExtModeMexFile	Not available	For MathWorks use only.
ExtModeTesting	Not available	For MathWorks use only.
FoldNonRolledExpr	Not available	For MathWorks use only.
GenerateFullHeader	Not available	For MathWorks use only.
IncAutoGenComments	Not available	For MathWorks use only.
IncludeRegionsInRTWFile BlockHierarchyMap	Not available	For MathWorks use only.
IncludeRootSignalInRTWFile	Not available	For MathWorks use only.
IncludeVirtualBlocksInRTW FileBlockHierarchyMap	Not available	For MathWorks use only.
IsERTTarget	Not available	For MathWorks use only.
ModelReferenceCompliant string - off, on	Not available	Set in SelectCallback for a target to indicate whether the target supports model reference.
ParamNamingFcn	Not available	For MathWorks use only.
PostCodeGenCommand string - ''	Not available	Add the specified post code generation command to the model build process.
PreserveName	Not available	For MathWorks use only.
PreserveNameWithParent	Not available	For MathWorks use only.
ProcessScript	Not available	For MathWorks use only.
ProcessScriptMode	Not available	For MathWorks use only.
SignalNamingFcn	Not available	For MathWorks use only.

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
SystemCodeInlineAuto	Not available	For MathWorks use only.
TargetFcnLib	Not available	For MathWorks use only.
TargetLibSuffix <i>string - ''</i>	Not available	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 string 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.) Note To use this parameter with the toolchain approach, see "Library Control Parameters"
TargetPreCompLibLocation <i>string - ''</i>	Not available	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> .
TargetPreprocMaxBitsSint <i>int - 32</i>	Not available	Specify the maximum number of bits that the target C preprocessor can use for signed integer math.
TargetPreprocMaxBitsUuint <i>int - 32</i>	Not available	Specify the maximum number of bits that the target C preprocessor can use for unsigned integer math.

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
TargetTypeEmulationWarn SuppressLevel SuppressLevel int - 0	Not available	When greater than or equal to 2, suppress warning messages that the Simulink Coder software displays when emulating integer sizes in rapid prototyping environments.
TLCOptions string - ''	Not available	Specify additional TLC command line options.

Model Advisor Checks

Simulink Coder Checks

In this section...
“Simulink® Coder™ Checks Overview” on page 5-3
“Identify blocks using one-based indexing” on page 5-4
“Check solver for code generation” on page 5-6
“Check for blocks not supported by code generation” on page 5-8
“Check and update model to use toolchain approach to build generated code” on page 5-9
“Check and update the embedded target model to use ert.tlc system target file” on page 5-12
“Check for blocks that have constraints on tunable parameters” on page 5-14
“Check for model reference configuration mismatch” on page 5-16
“Check sample times and tasking mode” on page 5-17
“Code Generation Advisor Checks” on page 5-17

Simulink Coder Checks Overview

Use Simulink Coder Model Advisor checks to configure your model for code generation.

See Also

- “Consult the Model Advisor”
- “Simulink Checks”
- “Embedded Coder Checks”

Identify blocks using one-based indexing

Identify blocks using one-based indexing.

Description

Zero-based indexing is more efficient in the generated code than one-based indexing.

Using zero-based indexing increases execution efficiency of the generated code.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks configured for one-based indexing.	Configure the blocks for zero-based indexing. Update the supporting blocks.
The model or subsystem contains one or more of the following, which require one-based indexing: <ul style="list-style-type: none"> • Fcn block • MATLAB functions inside Stateflow Charts • MATLAB Function block • MATLAB System block • Stateflow Charts with MATLAB action language • State Transition Table block • Truth Table block 	Evaluate the blocks to determine if one-based indexing is used. Consider replacing the blocks with Simulink basic blocks.

See Also

“cgsl_0101: Zero-based indexing”.

Capabilities and Limitations

You can run this check on your library models.

Check solver for code generation

Check model solver and sample time configuration settings.

Description

Incorrect configuration settings can stop the Simulink Coder software from generating code. Underspecifying sample times can lead to undesired results. Avoid generating code that might corrupt data or produce unpredictable behavior.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The solver type is set incorrectly for model level code generation.	In the Configuration Parameters dialog box, on the Solver pane, set <ul style="list-style-type: none"> • Type to Fixed-step • Solver to Discrete (no continuous states)
Multitasking diagnostic options are not set to error.	In the Configuration Parameters dialog box, on the Diagnostics pane, set <ul style="list-style-type: none"> • Sample Time > Multitask conditionally executed subsystem to error • Sample Time > Multitask rate transition to error • Data Validity > Multitask data store to error

Tips

You do not have to modify the solver settings to generate code from a subsystem. The Embedded Coder build process automatically changes **Solver**

type to fixed-step when you select **Code Generation > Build Subsystem** or **Code Generation > Generate S-Function** from the subsystem context menu.

See Also

- “Configure Scheduling”
- “Execute Multitasking Models”

Check for blocks not supported by code generation

Identify blocks not supported by code generation.

Description

This check partially identifies model constructs that are not suited for code generation as identified in the Simulink Block Support tables for Simulink Coder and Embedded Coder. If you are using blocks with support notes for code generation, review the information and follow the given advice.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for code generation.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.

Capabilities and Limitations

You can run this check on your library models.

See Also

“Supported Products and Block Usage”

Check and update model to use toolchain approach to build generated code

Check if model uses Toolchain settings to build generated code.

Description

Checks whether the model uses the template makefile approach or the toolchain approach to build the generated code.

Available with Simulink Coder.

When you open a model created before R2013b that has **System target file** set to `ert.tlc`, `ert_shrlib.tlc`, or `grt.tlc` the software automatically tries to upgrade the model from using the template makefile approach to using the toolchain approach.

If the software did not upgrade the model, this check determines the cause, and if available, recommends actions you can perform to upgrade the model.

To determine which approach your model is using, you can also look at the Code Generation pane in the Configuration Parameters dialog box. The toolchain approach uses the following parameters to build generated code:

- “Toolchain” on page 4-14
- “Build configuration” on page 4-16

The template makefile approach uses the following settings to build generated code:

- **Compiler optimization level**
- **Custom compiler optimization flags**
- **Generate makefile**
- **Template makefile**

Results and Recommended Actions

Condition	Recommended Action	Comment
Model is configured to use the toolchain approach.	No action.	The model was automatically upgraded.
Model is not configured to use the toolchain approach.	Model cannot be automatically upgraded to use the toolchain approach.	The system target file is not toolchain-compliant. Set System target file to a toolchain-compliant target, such as <code>ert.tlc</code> , <code>ert_shrllib.tlc</code> , or <code>grt.tlc</code> .
Model is not configured to use the toolchain approach. (Parameter values are not the default values.)	Model can be automatically upgraded to use the toolchain approach. Click Update Model .	The parameters are set to their default values, except Compiler Optimization Level , which is set <code>Optimizations on</code> (faster runs). Clicking Update Model sets Compiler Optimization Level to its default value, <code>Optimizations off</code> (faster builds), and then upgrades the model. The upgraded model has Build Configuration set to <code>Faster Builds</code> . Saving the model makes these changes permanent.
Model is not configured to use the toolchain approach. (Parameter values are not the default values.)	Model cannot be automatically upgraded to use the toolchain approach.	One or more of the following parameters is not set to its default value: <ul style="list-style-type: none"> • Generate makefile (default: Enabled) • Template makefile (default: Target-specific default TMF) • Compiler optimization level (default: <code>Optimizations off</code> (faster builds))

Condition	Recommended Action	Comment
		<ul style="list-style-type: none">• Make command (default: <code>make_rtw</code> without arguments) See “Upgrade Model to Use Toolchain Approach”

Action Results

Clicking **Update model** upgrades the model to use the toolchain approach to build generated code.

See Also

- “Upgrade Model to Use Toolchain Approach”

Check and update the embedded target model to use ert.tlc system target file

Check and update the embedded target model to use ert.tlc system target file.

Description

Check and update models whose **System target file** is set to `idelink_ert.tlc` or `idelink_grt.tlc` and whose target hardware is one of the supported Texas Instruments C2000 processors to use `ert.tlc` and similar settings.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
System target file is set to <code>ert.tlc</code> - Embedded Coder.	No action
System target file is set to <code>idelink_ert.tlc</code> or <code>idelink_grt.tlc</code> and Board parameter is set to a processor that is supported by the Embedded Coder Support Package for Texas Instruments C2000 Processors.	Update model

Action Results

Clicking **Update model** automatically sets the following parameters on the **Code Generation** pane in the model Configuration Parameters dialog box:

- **System target file** parameter to `ert.tlc`.
- **Target hardware** parameter to match the previous board or processor.
- **Toolchain** parameter to match the previous toolchain.
- **Build configuration** parameter to match the build configuration.

This action also sets the parameters on the **Coder Target** pane to match the previous parameter values under the **Peripherals** tab.

Capabilities and Limitations

The new workflow uses the toolchain approach, which relies on enhanced makefiles to build generated code. It does not provide an equivalent to setting the **Build format** parameter to **Project** in the previous configuration. Therefore, the new workflow cannot automatically generate IDE projects within the CCS 3.3 IDE.

See Also

- “Coder Target Pane: Texas Instruments C2000 Processors”
- “Toolchain”

Check for blocks that have constraints on tunable parameters

Identify blocks with constraints on tunable parameters.

Description

Lookup Table blocks have strict constraints when they are tunable. If you violate lookup table block restrictions, the generated code produces wrong answers.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: <ul style="list-style-type: none"> • Preserve monotonicity of the setting for the Vector of input values parameter. • Preserve the number and location of zero values that you specify for Vector of input values and Vector of output values parameters if you specify multiple zero values for the Vector of input values parameter.
Lookup Table (2-D) blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: <ul style="list-style-type: none"> • Preserve monotonicity of the setting for the Row index input values and Column index of input values parameters. • Preserve the number and location of zero values that you specify

Condition	Recommended Action
	for Row index input values , Column index of input values , and Vector of output values parameters if you specify multiple zero values for the Row index input values or Column index of input values parameters.
Lookup Table (n-D) blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must preserve the increasing monotonicity of the breakpoint values for each table dimension Breakpoints n .

See Also

- 1-D Lookup Table
- 2-D Lookup Table

Check for model reference configuration mismatch

Identify referenced model configuration parameter settings that do not match the top model configuration parameter settings.

Description

The code generator cannot create code for top models that contain referenced models with different, incompatible configuration parameter settings.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The top model and the referenced model have inconsistent model configuration parameter settings.	Modify the specified model configuration settings.

See Also

Model Referencing Configuration Parameter Requirements

Check sample times and tasking mode

Set up the sample time and tasking mode for your system.

Description

Incorrect tasking mode can result in inefficient code execution or incorrect generated code.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model represents a multirate system but is not configured for multitasking.	In the Configuration Parameters dialog box, on the Solver pane, set the Tasking mode for periodic sample times parameter as recommended.
The model is configured for multitasking, but multitasking is not desirable on the target hardware.	In the Configuration Parameters dialog box, on the Solver pane, set the Tasking mode for periodic sample times parameter to SingleTasking , or change the settings on the Hardware Implementation pane.

See Also

“Single-Tasking and Multitasking Execution Modes”

Code Generation Advisor Checks

- “Available Checks for Code Generation Objectives” on page 5-18
- “Identify questionable blocks within the specified system” on page 5-24
- “Check model configuration settings against code generation objectives” on page 5-25

Available Checks for Code Generation Objectives

Code generation objectives checks facilitate designing and troubleshooting Simulink models and subsystems that you want to use to generate code.

The Code Generation Advisor includes the following checks for each of the code generation objectives.

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check model configuration settings against code generation objectives” on page 5-25	Included	Included	Included	Included	Included	Included	Included	Included
“Check for optimal bus virtuality”	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
“Identify questionable blocks within the specified system” on page 5-24	Included	Included	Included	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check the hardware implementation”	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable software environment specifications”	Included when Traceability is not a higher priority and Embedded Coder is available	Included when Traceability is not a higher priority and Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable code instrumentation (data I/O)”	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	N/A	N/A	N/A	N/A	N/A
“Identify questionable subsystem settings”	N/A	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify blocks that generate expensive rounding code”	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify questionable fixed-point operations”	Included if Embedded Coder or Fixed-Point Designer is available	Included if Embedded Coder or Fixed-Point Designer is available	N/A	N/A	N/A	N/A	N/A	N/A
“Identify blocks using one-based indexing” on page 5-4	Included	Included	N/A	N/A	N/A	N/A	N/A	N/A
“Identify lookup table blocks that generate expensive out-of-range checking code”	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check output types of logic blocks”	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A	N/A
“Identify unconnected lines, input ports, and output ports”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify block output signals with continuous sample time and non-floating point data type”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for blocks that have constraints on tunable parameters” on page 5-14	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check if read/write diagnostics are enabled for data store blocks”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for partial structure parameter usage with bus signals”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check data store block sample times for modeling errors”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for potential ordering issues involving data store access”	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for blocks not recommended for	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precaution (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
MISRA-C:2004 compliance								

See Also.

- “Application Objectives” in the Simulink Coder documentation.
- “Application Objectives” in the Embedded Coder documentation.
- “Consult the Model Advisor” in the Simulink documentation.
- Simulink Model Advisor Check Reference in the Simulink documentation.
- “Simulink® Coder™ Checks” on page 5-2.
- Simulink Verification and Validation Model Advisor Check Reference in the Simulink Verification and Validation documentation.

Identify questionable blocks within the specified system

Identify blocks not supported by code generation or not recommended for deployment.

Description. The code generator creates code only for the blocks that it supports. Some blocks are not recommended for production code deployment.

Results and Recommended Actions.

Condition	Recommended Action
A block is not supported by the Simulink Coder software.	Remove the specified block from the model or replace the block with the recommended block.
A block is not recommended for production code deployment.	Remove the specified block from the model or replace the block with the recommended block.
Check for Gain blocks whose value equals 1.	Replace Gain blocks with Signal Conversion blocks.

Capabilities and Limitations. You can run this check on your library models.

See Also. “Supported Products and Block Usage”

Check model configuration settings against code generation objectives

Check the configuration parameter settings for the model against the code generation objectives.

Description. Each parameter in the Configuration Parameters dialog box might have different recommended settings for code generation based on your objectives. This check helps you identify the recommended setting for each parameter so that you can achieve optimized code based on your objective.

Results and Recommended Actions.

Condition	Recommended Action
Parameters are set to values other than the value recommended for the specified objectives.	Set the parameters to the recommended values. <hr/> Note A change to one parameter value can impact other parameters. Passing the check might take multiple iterations. <hr/>

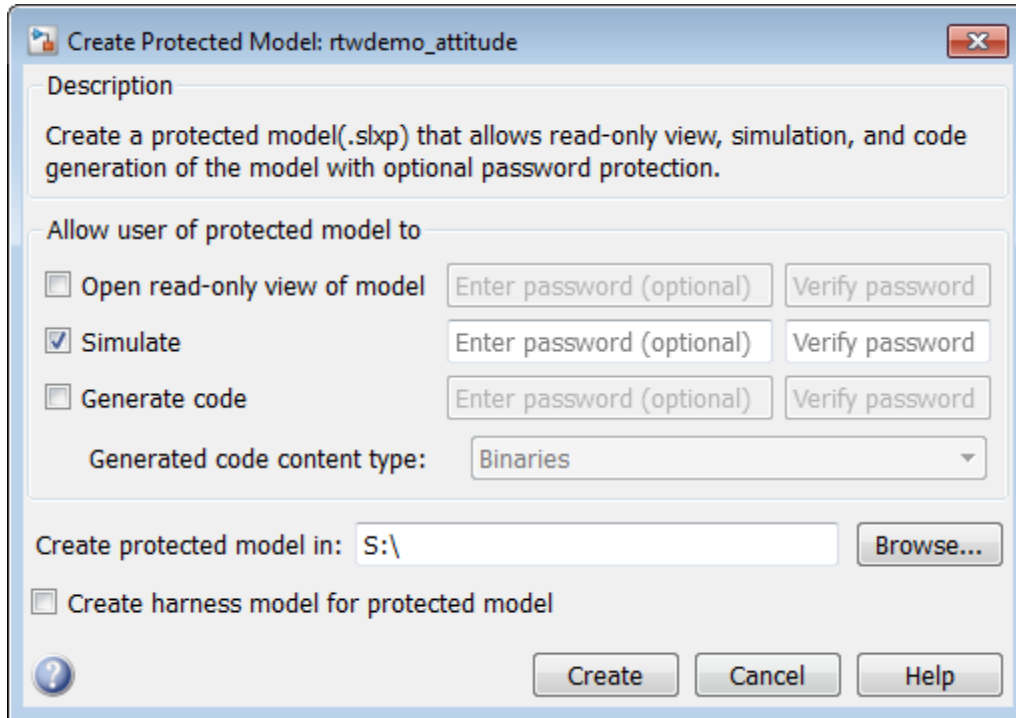
Action Results. Clicking **Modify Parameters** changes the parameter values to the recommended values.

See Also.

- The Simulink Coder “Recommended Settings Summary” on page 4-345
- The Embedded Coder “Recommended Settings Summary”
- “Application Objectives” in the Simulink Coder documentation.
- “Application Objectives” in the Embedded Coder documentation.

Parameters for Creating Protected Models

Create Protected Model



In this section...

“Create Protected Model: Overview” on page 6-3

“Open read-only view of model” on page 6-4

“Simulate” on page 6-5

“Generate code” on page 6-6

“Generated code content type” on page 6-7

“Create protected model in” on page 6-7

“Create harness model for protected model” on page 6-9

Create Protected Model: Overview

Create a protected model (.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.

To open the Create Protected Model dialog box, right-click the model block that references the model for which you want to generate protected model code. From the context menu, select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**.

See Also

- “Protected Model”
- “Create a Protected Model”

Open read-only view of model

Share a view-only version of your protected model with optional password protection. View-only version includes the contents and block parameters of the model.

Settings

Default: Off



On

Share a Web view of the protected model. For password protection, create and verify a password with a minimum of four characters.



Off

Do not share a Web view of the protected model.

Dependencies

- The protected model user must have a Report Generator license to use the model Web view.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Create a Protected Model”
- “Protect a Referenced Model”

Simulate

Allow user to simulate a protected model with optional password protection.

Selecting **Simulate**:

- Enables protected model Simulation Report.
- Sets Mode to Accelerator. You can run Normal Mode and Accelerator simulations.
- Displays only binaries and headers.
- Enables code obfuscation.

Settings

Default: On



On

User can simulate the protected model. For password protection, create and verify a password with a minimum of four characters.



Off

User cannot simulate the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Create a Protected Model”
- “Protect a Referenced Model”

Generate code

Allows user to generate code for the protected model with optional password protection. Selecting **Generate code**:

- Enables Simulation Report and Code Generation Report for the protected model.
- Sets Mode to enable code generation.
- Enables support for simulation.
- Displays code in the build folder in obfuscated form.

Settings

Default: Off



User can generate code for the protected model. For password protection, create and verify a password with a minimum of four characters.



User cannot generate code for the protected model.

Dependencies

- To generate code, you must also select the **Simulate** check box.
- This parameter enables **Generated code content type**.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Support in a Protected Model”
- “Protect a Referenced Model”

Generated code content type

Select the appearance of the generated code. When you select the **Generate code** check box, this parameter is enabled.

Settings

Default: Obfuscated source code

Binaries

Includes only binaries for the generated code.

Obfuscated source code

Includes obfuscated headers and binaries for the generated code.

Readable source code

Includes readable source code.

Dependencies

This parameter is enabled by selecting the **Generate code** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Support in a Protected Model”
- “Protect a Referenced Model”

Create protected model in

Specify the folder path for the protected model.

Settings

Default: Current working folder

Dependencies

A model that you protect must be available on the MATLAB path.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect a Referenced Model”
- “Create a Protected Model”

Create harness model for protected model

Create a harness model for the protected model. The harness model contains only a Model block that references the protected model.

Settings

Default: Off



On

Create a harness model for the protected model.



Off

Do not create a harness model for the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Harness Model”
- “Test the Protected Model”