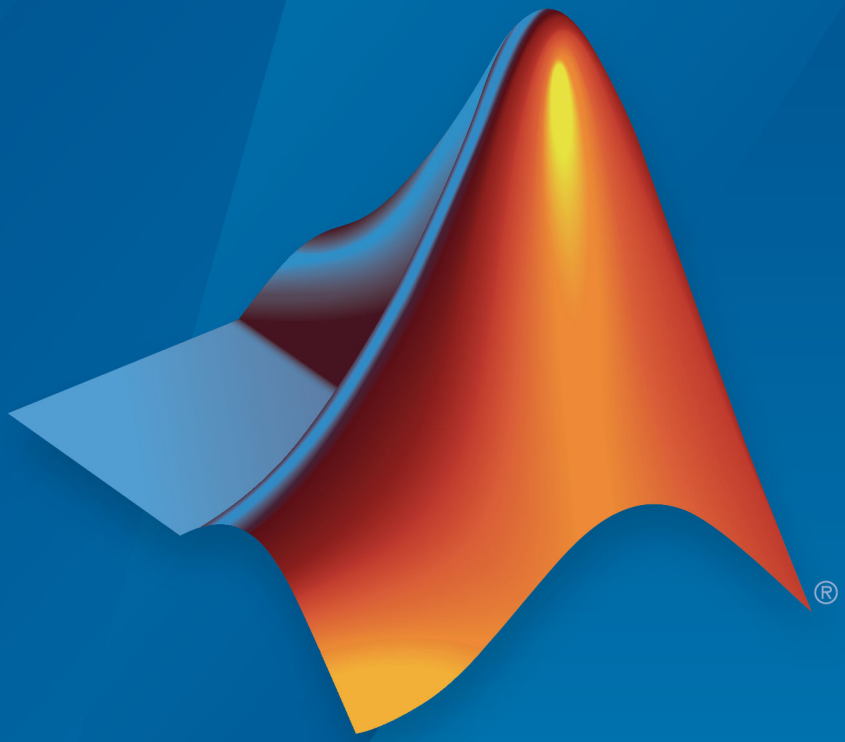
Simulink® Coder™

Reference



MATLAB® & SIMULINK®

R2018b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

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

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

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

Phone: 508-647-7000

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

Simulink® Coder™ Reference

Revision History

April 2011	Online only	New for Version 8.0 (Release 2011a)
September 2011	Online only	Revised for Version 8.1 (Release 2011b)
March 2012	Online only	Revised for Version 8.2 (Release 2012a)
September 2012	Online only	Revised for Version 8.3 (Release 2012b)
March 2013	Online only	Revised for Version 8.4 (Release 2013a)
September 2013	Online only	Revised for Version 8.5 (Release 2013b)
March 2014	Online only	Revised for Version 8.6 (Release 2014a)
October 2014	Online only	Revised for Version 8.7 (Release 2014b)
March 2015	Online only	Revised for Version 8.8 (Release 2015a)
September 2015	Online only	Revised for Version 8.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 8.10 (Release 2016a)
September 2016	Online only	Revised for Version 8.11 (Release 2016b)
March 2017	Online only	Revised for Version 8.12 (Release 2017a)
September 2017	Online only	Revised for Version 8.13 (Release 2017b)
March 2018	Online only	Revised for Version 8.14 (Release 2018a)
September 2018	Online only	Revised for Version 9.0 (Release 2018b)

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.

Contents

1 Simulink Code Generation Limitations

2 Functions in Simulink Coder—Alphabetical List

3 Blocks in Simulink Coder—Alphabetical List

4 Code Generation Parameters: Code Generation

5 Code Generation Parameters: Report

6 Code Generation Parameters: Comments

7 Code Generation Parameters: Symbols

8 Code Generation Parameters: Custom Code

9 Code Generation Parameters: Interface

10 Simulink Coder Parameters: Advanced Parameters

11 Configuration Parameters for Simulink Models

12 Model Advisor Checks

13 Parameters for Creating Protected Models

14 Tools in Simulink Coder—Alphabetical List

15 Optimization Parameters

1

Simulink Code Generation Limitations

Simulink Code Generation Limitations

The following topics identify Simulink code generation limitations:

- “C++ Language Support Limitations”
- “packNGo Function Limitations”
- “Tunable Expression Limitations”
- “Generate Reentrant Code from Subsystems”
- “Simulink Coder Model Referencing Limitations”
- “TCP/IP and Serial External Mode Limitations”
- “Noninlined S-Function Parameter Type Limitations”
- “S-Function Target Limitations”
- “Rapid Simulation Target Limitations”
- “Asynchronous Support Limitations”
- “C API Limitations”
- “Blocks and Products Supported for C Code Generation”

Functions in Simulink Coder—Alphabetical List

addCompileFlags

Add compiler options to model build information

Syntax

```
addCompileFlags(buildinfo,options,groups)
```

Description

`addCompileFlags(buildinfo,options,groups)` specifies the compiler options to add to the build information.

The function requires the *`buildinfo`* and *`options`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the compiler options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Compiler Flags to OPTS Group

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

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

Add Compiler Flags to OPT_OPTS Group

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

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

Add Compiler Flags to Build Information

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

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`options` — List of compiler options to add to build information

character vector | array of character vectors | string

You can specify the *`options`* argument as a character vector, as an array of character vectors, or as a string You can specify the *`options`* argument as multiple compiler flags within a single character vector, for example `'-Zi -Wall'`. If you specify the *`options`* argument as multiple character vectors, for example, `'-Zi -Wall'` and `'-O3'`, the *`options`* argument is added to the build information as an array of character vectors.

Example: `{'-Zi -Wall' '-O3'}`

`groups` — Optional group name for the added compiler options

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'Debug' 'MemOpt'`, the function relates the *`groups`* to the *`options`* in order of appearance. For example, the *`options`* argument `{'-Zi -Wall' '-O3'}` is an array of character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `'MemOpt'` group.

Example: `{'Debug' 'MemOpt'}`

See Also

`addDefines` | `addLinkFlags` | `getCompileFlags`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addDefines

Add preprocessor macro definitions to model build information

Syntax

```
addDefines(buildinfo,macrodefs,groups)
```

Description

`addDefines(buildinfo,macrodefs,groups)` specifies the preprocessor macro definitions to add to the build information.

The function requires the *`buildinfo`* and *`macrodefs`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the definitions in a build information object. The function adds definitions to the object based on the order in which you specify them.

Examples

Add Macro Definitions to OPTS Group

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

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

Add Macro Definitions to OPT_OPTS Group

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

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

Add Macro Definitions to Build Information

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

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`macrodefs` — List of macro definitions to add to build information

character vector | array of character vectors | string

You can specify the *`macrodefs`* argument as a character vector, as an array of character vectors, or as a string. You can specify the *`macrodefs`* argument as multiple definitions within a single character vector, for example `'-DRT -DDEBUG'`. If you specify the *`macrodefs`* argument as multiple character vectors, for example`'-DPROTO -DDEBUG'` and `'-DPRODUCTION'`, the *`macrodefs`* argument is added to the build information as an array of character vectors.

Example: `{'-DPROTO -DDEBUG' '-DPRODUCTION'}`

`groups` — Optional group name for the added compiler options

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example `'Debug' 'Release'`, the function relates the *`groups`* to the *`macrodefs`* in order of appearance. For example, the *`macrodefs`* argument `{'-DPROTO -DDEBUG' '-DPRODUCTION'}` is an array of

character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `'Release'` group.

Example: `{'Debug' 'Release'}`

See Also

`addCompileFlags` | `addLinkFlags` | `getDefines`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addIncludeFiles

Add include files to model build information

Syntax

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

Description

`addIncludeFiles(buildinfo,filenames,paths,groups)` specifies included files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`paths`* argument to specify the included file paths and use an optional *`groups`* argument to group your options.

The code generator stores the included file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Included File to SysFiles Group

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

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

Add Included Files to AppFiles Group

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

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

Add Included Files to SysFiles and AppFiles Groups

Add the include files `etc.h`, `etc_private.h`, and `mytypes.h` to the build information `myModelBuildInfo`. Group the files `etc.h` and `etc_private.h` with the character vector `AppFiles` and the file `mytypes.h` with the character vector `SysFiles`.

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

Add Included Files with Wildcard to HFiles Group

Add the include files (`.h` files identified with a wildcard character) in a specified folder to the build information `myModelBuildInfo`, and place the files in the group `HFiles`.

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of included files to add to build information
character vector | array of character vectors | string

You can specify the *filenames* argument as a character vector, as an array of character vectors, or as a string. If you specify the *filenames* argument as multiple character vectors, for example, `'etc.h' 'etc_private.h'`, the *filenames* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.h'`, and `'*.h*'`.

The function removes duplicate included file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.h'`

paths — List of included file paths to add to build information
character vector | array of character vectors | string

You can specify the *paths* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *paths* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *paths* argument is added to the build information as an array of character vectors.

Example: `'/proj/src'`

groups — Optional group name for the added included files
character vector | array of character vectors | string

You can specify the *groups* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *groups*, for example, `'AppFiles' 'AppFiles' 'SysFiles'`, the function relates the *groups* to the *filenames* in order of appearance. For example, the *filenames* argument `'etc.h' 'etc_private.h' 'mytypes.h'` is an array of character vectors with three elements. The first element is in the `'AppFiles'` group, the second element is in the `'AppFiles'` group, and the third element is in the `'SysFiles'` group.

Example: `'AppFiles' 'AppFiles' 'SysFiles'`

See Also

`addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `findIncludeFiles` | `getIncludeFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addIncludePaths

Add include paths to model build information

Syntax

```
addIncludePaths(buildinfo,paths,groups)
```

Description

`addIncludePaths(buildinfo,paths,groups)` specifies included file paths to add to the build information.

The function requires the *`buildinfo`* and *`paths`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the included file path options in a build information object. The function adds options to the object based on the order in which you specify them.

The code generator does not check whether a specified path is valid.

Examples

Add Include File Path to Build Information

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

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

Add Include File Paths to a Group

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

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

Add Include File Paths to Groups

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

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`paths` — List of included file paths to add to build information

character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

The function removes duplicate include file path entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'/proj/src'`

`groups` — Optional group name for the added included files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'etc' 'etc' 'shared'`, the function relates the *`groups`* to the *`paths`* in order of appearance. For example, the *`paths`* argument `'/etc/proj/etclib' '/etcproj/etc/etc_build' '/common/lib'` is an array of character vectors with three elements. The first element is in the `'etc'` group, the second element is in the `'etc'` group, and the third element is in the `'shared'` group.

Example: `'etc' 'etc' 'shared'`

See Also

`addIncludeFiles` | `addSourceFiles` | `addSourcePaths` | `getIncludePaths` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addLinkFlags

Add link options to model build information

Syntax

```
addLinkFlags(buildinfo,options,groups)
```

Description

`addLinkFlags(buildinfo,options,groups)` specifies the linker options to add to the build information.

The function requires the *`buildinfo`* and *`options`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the linker options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Linker Flags to OPTS Group

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

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

Add Linker Flags to OPT_OPTS Group

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

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

Add Linker Flags to Build Information

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

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`options` — List of linker options to add to build information

character vector | array of character vectors | string

You can specify the *`options`* argument as a character vector, as an array of character vectors, or as a string. You can specify the *`options`* argument as multiple compiler flags within a single character vector, for example `'-MD -Gy'`. If you specify the *`options`* argument as multiple character vectors, for example, `'-MD -Gy'` and `'-T'`, the *`options`* argument is added to the build information as an array of character vectors.

Example: `{'-MD -Gy' '-T'}`

`groups` — Optional group name for the added linker options

character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'Debug' 'Temp'`, the function relates the *`groups`* to the *`options`* in order of appearance. For example, the *`options`* argument `{'-MD -Gy' '-T'}` is an array of character vectors with two elements. The first element is in the `'Debug'` group and the second element is in the `Temp'` group.

Example: `{'Debug' 'Temp'}`

See Also

`addCompileFlags` | `addDefines` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addLinkObjects

Add link objects to model build information

Syntax

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

Description

`addLinkObjects(buildinfo,linkobjs,paths,priority,precompiled, linkonly,groups)` specifies included files and paths to add to the build information.

The function requires the *`buildinfo`*, *`linkobs`*, and *`paths`* arguments. You can optionally select *`priority`* for link objects, select whether the objects are *`precompiled`*, select whether the objects are *`linkonly`* objects, and apply a *`groups`* argument to group your options.

The code generator stores the included link object and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Link Objects to Build Information

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

```
myModelBuildInfo = RTW.BuildInfo;
addLinkObjects(myModelBuildInfo,{'libobj1' 'libobj2'}, ...
```

```
   {'/proj/lib/lib1' '/proj/lib/lib2'},1000, ...
   false,true);
```

Add Prioritized Link-Only Link Objects to Build Information

Add the linkable objects `libobj1` and `libobj2` to the build information `myModelBuildInfo`. Set the priorities of the objects to 26 and 10, respectively. Because `libobj2` is assigned the lower numeric priority value and has the higher priority, the function orders the objects such that `libobj2` precedes `libobj1` in the vector.

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

Add Precompiled Link Objects to MyTest Group

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

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`

object

`linkobjs` — List of linkable object files to add to build information

character vector | array of character vectors | string

You can specify the *`linkobjs`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`linkobjs`* argument as multiple character vectors, for example, `'libobj1' 'libobj2'`, the *`linkobjs`* argument is added to the build information as an array of character vectors.

The function removes duplicate linkable object entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'libobj1'`

`paths` — List of included file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/lib/lib1'` and `'/proj/lib/lib2'`, the *`paths`* argument is added to the build information as an array of character vectors. The number of elements in *`paths`* must match the number of elements in the `linkobjs` argument.

Example: `'/proj/lib/lib1'`

`priority` — List of priority values for link objects to add to build information
`1000` (default) | numeric value | array of numeric values

A numeric value or an array of numeric values that indicates the relative priority of each specified link object. Lower values have higher priority.

Example: `1000`

`precompiled` — List of precompiled indicators for link objects to add to build information
false (default) | true | array of logical values

A logical value or an array of logical values that indicates whether each specified link object is precompiled. The logical value `true` indicates precompiled.

Example: `false`

`linkonly` — List of link-only indicators for link objects to add to build information
`false` (default) | true

A logical value or an array of logical values that indicates whether each specified link object is link-only (no precompilation). The logical value `true` indicates link-only. If *`linkonly`* is `true`, the value of the *`precompiled`* argument is ignored.

Example: `false`

`groups` — Optional group name for the added link object files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'MyTest1' 'MyTest2'`, the function relates the *`groups`* to the *`linkobjs`* in order of appearance. For example, the *`linkobjs`* argument `'libobj1' 'libobj2'` is an array of character vectors with two elements. The first element is in the `'MyTest1'` group, and the second element is in the `'MyTest2'` group.

Example: `'MyTest1' 'MyTest2'`

See Also

`addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `findIncludeFiles` | `getIncludeFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addNonBuildFiles

Add nonbuild-related files to model build information

Syntax

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

Description

`addNonBuildFiles(buildinfo,filenames,paths,groups)` specifies nonbuild-related files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`paths`* argument to specify the included file paths and use an optional *`groups`* argument to group your options.

The code generator stores the nonbuild-related file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Nonbuild File to DocFiles Group

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

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

Add Nonbuild Files to DLLFiles Group

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

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

Add Nonbuild Files with Wildcard to DLLFiles Group

Add nonbuild-related files (`.dll` files identified with a wildcard character) in a specified folder to the build information `myModelBuildInfo`, and place the files in the group `DLLFiles`.

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of nonbuild-related files to add to build information
character vector | array of character vectors | string

You can specify the *`filenames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`filenames`* argument as multiple character vectors, for example, `'etc.dll' 'etc_private.dll'`, the *`filenames`* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.dll'`, and `'*.d*'`.

The function removes duplicate nonbuild-related file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.dll'`

`paths` — List of nonbuild-related file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/dll'` and `'/proj/docs'`, the *`paths`* argument is added to the build information as an array of character vectors.

Example: `'/proj/dll'`

`groups` — Optional group name for the added nonbuild-related files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'DLLFiles' 'DLLFiles' 'DocFiles'`, the function relates the *`groups`* to the *`filenames`* in order of appearance. For example, the *`filenames`* argument `''myutility1.dll'' ''myutility2.dll'' 'readme.txt'` is an array of character vectors with three elements. The first element is in the `'DLLFiles'` group, the second element is in the `'DLLFiles'` group, and the third element is in the `'DocFiles'` group.

Example: `'DLLFiles' 'DLLFiles' 'DocFiles'`

See Also

`getNonBuildFiles`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2008a

addSourceFiles

Add source files to model build information

Syntax

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

Description

`addSourceFiles(buildinfo,filenames,paths,groups)` specifies source files and paths to add to the build information.

The function requires the *`buildinfo`* and *`filenames`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the source file and path options in a build information object. The function adds options to the object based on the order in which you specify them.

Examples

Add Source File to Drivers Group

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

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

Add Source Files to a Group

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

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

Add Source Files to Groups

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

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

Add Source Files with Wildcard to CFiles Group

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

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`filenames` — List of source files to add to build information
character vector | array of character vectors | string

You can specify the *`filenames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`filenames`* argument as multiple character vectors, for example, `'etc.c' 'etc_private.c'`, the *`filenames`* argument is added to the build information as an array of character vectors.

If the dot delimiter (`.`) is present, the file name text can include wildcard characters. Examples are `'*.*'`, `'*.c'`, and `'*.c*'`.

The function removes duplicate included file entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'*.c'`

`paths` — List of source file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

Example: `'/proj/src'`

`groups` — Optional group name for the added source files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'Tests' 'Tests' 'Drivers'`, the function relates the *`groups`* to the *`filenames`* in order of appearance. For example, the *`filenames`* argument `'test1.c' 'test2.c' 'driver.c'` is an array of character vectors with three elements. The first element is in the `'Tests'` group, and the second element is in the `'Tests'` group, and the third element is in the `'Drivers'` group.

Example: `'Tests' 'Tests' 'Drivers'`

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourcePaths` | `getSourceFiles` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addSourcePaths

Add source paths to model build information

Syntax

```
addSourcePaths(buildinfo,paths,groups)
```

Description

`addSourcePaths(buildinfo,paths,groups)` specifies source file paths to add to the build information.

The function requires the *`buildinfo`* and *`paths`* arguments. You can use an optional *`groups`* argument to group your options.

The code generator stores the source file path options in a build information object. The function adds options to the object based on the order in which you specify them.

The code generator does not check whether a specified path is valid.

Examples

Add Source File Path to Build Information

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

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

Add Source File Paths to a Group

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

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

Add Source File Paths to Groups

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

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`paths` — List of source file paths to add to build information
character vector | array of character vectors | string

You can specify the *`paths`* argument as a character vector, as an array of character vectors, or as a string. If you specify a single path as a character vector, the function uses that path for all files. If you specify the *`paths`* argument as multiple character vectors, for example, `'/proj/src'` and `'/proj/inc'`, the *`paths`* argument is added to the build information as an array of character vectors.

The function removes duplicate source file path entries with an exact match of a path and file name to a previously defined entry in the build information object.

Example: `'/proj/src'`

`groups` — Optional group name for the added source files
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'etc' 'etc' 'shared'`, the function relates the *`groups`* to the *`paths`* in order of appearance. For example, the *`paths`* argument `'/etc/proj/etclib' '/etcproj/etc/etc_build' '/common/lib'` is an array of character vectors with three elements. The first element is in the `'etc'` group, the second element is in the `'etc'` group, and the third element is in the `'shared'` group.

Example: `'etc' 'etc' 'shared'`

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `getSourcePaths` | `updateFilePathsAndExtensions` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

addTMFTokens

Add template makefile (TMF) tokens to model build information

Syntax

```
addTMFTokens(buildinfo,tokennames,tokenvalues,groups)
```

Description

`addTMFTokens(buildinfo,tokennames,tokenvalues,groups)` specifies TMF tokens and values to add to the build information.

To provide build-time information to help customize makefile generation, call the `addTMFTokens` function inside a post-code-generation command. The tokens specified in the `addTMFTokens` function call must be handled in the template makefile (TMF) for the target selected for your model. For example, you can call `addTMFTokens` in a post-code-generation command to add a TMF token named `|>CUSTOM_OUTNAME<|` with a token value that specifies an output file name for the build. To achieve the result you want, the TMF must apply an action with the value of `|>CUSTOM_OUTNAME<|`. (See “Examples” on page 2-0 .)

The `addTMFTokens` function adds specified TMF token names and values to the model build information. The code generator stores the TMF tokens in a vector. The function adds the tokens to the end of the vector in the order that you specify them.

The function requires the *`buildinfo`*, *`tokennames`*, and *`tokenvalues`* arguments. You can use an optional *`groups`* argument to group your options. You can specify *`groups`* as a character vector or as an array of character vectors.

Examples

Add TMF Tokens to Build Information

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

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

Apply Build Information as Tokens in TMF Build

In the TMF for the target selected for your model, this code uses the token value to achieve the result that you want:

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

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`tokennames` — Specifies names of TMF tokens to add to the build information
character vector | array of character vectors | string

You can specify the *`tokennames`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`tokennames`* argument as multiple character vectors, for example, `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'`, the *`tokennames`* argument is added to the build information as an array of character vectors.

Example: `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'`

`tokenvalues` — Specifies TMF token values (for the added tokens) to add to the build information
character vector | array of character vectors | string

You can specify the *`tokenvalues`* argument as a character vector, as an array of character vectors, or as a string. If you specify the *`tokenvalues`* argument as multiple

character vectors, for example, `'|>CUSTOM_OUTNAME<|' 'PCWIN64'`, the *`tokennames`* argument is added to the build information as an array of character vectors.

Example: `'foo.exe' 'PCWIN64'`

`groups` — Optional group name for the added TMF tokens
character vector | array of character vectors | string

You can specify the *`groups`* argument as a character vector, as an array of character vectors, or as a string. If you specify multiple *`groups`*, for example, `'LINK_INFO' 'COMPUTER_INFO'`, the function relates the *`groups`* to the *`tokennames`* in order of appearance. For example, the *`tokennames`* argument `'|>CUSTOM_OUTNAME<|' '|>COMPUTER<|'` is an array of character vectors with two elements. The first element is in the `'LINK_INFO'` group, and the second element is in the `'COMPUTER_INFO'` group.

Example: `'LINK_INFO' 'COMPUTER_INFO'`

See Also

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2009b

buildStandaloneCoderAssumptions

Create application to check code generator assumptions

Syntax

```
buildStandaloneCoderAssumptions(buildFolder)
```

Description

`buildStandaloneCoderAssumptions(buildFolder)` creates an application for your target hardware to check code generator assumptions. The application checks that code generator assumptions based on model parameter settings or build configuration settings are correct with reference to the target hardware.

The function creates the target application in the `buildFolder\coderassumptions\standalone` subfolder.

Examples

Create Application to Check Code Generator Assumptions

For an example that shows how to create an application to check code generator assumptions, see “Check Code Generator Assumptions for Development Computer” (Embedded Coder).

Input Arguments

`buildFolder` — Build folder
character vector | string scalar

Path to the build folder that contains the generated code.

See Also

Topics

“Check Code Generation Assumptions” (Embedded Coder)

Introduced in R2018b

coder.buildstatus.close

Close build process status window

Syntax

```
coder.buildstatus.close()
coder.buildstatus.close(model)
coder.buildstatus.close(subsystem)
```

Description

`coder.buildstatus.close()` closes any open **Build Process Status** windows.

The **Build Process Status** window supports parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

`coder.buildstatus.close(model)` closes the **Build Process Status** window for the `model`.

`coder.buildstatus.close(subsystem)` closes the **Build Process Status** window for the `subsystem`.

Examples

Close Build Process Status Windows

Close any open **Build Process Status** windows.

```
coder.buildstatus.close()
```

Close Build Process Status Window for a Model

After generating code for `rtwdemo_counter`, close the **Build Process Status** window for the model.

```
coder.buildstatus.close('rtwdemo_counter')
```

Close Build Process Status Window for a Subsystem

Close the **Build Process Status** window for the subsystem 'Amplifier' in model 'rtwdemo_counter'.

```
coder.buildstatus.close('rtwdemo_counter/Amplifier')
```

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: 'rtwdemo_counter'

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: 'rtwdemo_counter/Amplifier'

Data Types: `char`

See Also

`coder.buildstatus.open` | `coder.report.close` | `rtwbuild` | `slbuild`

Topics

"View Build Process Status"

Introduced in R2018a

coder.buildstatus.open

Open build process status window

Syntax

```
coder.buildstatus.open(model)
coder.buildstatus.open(model,systemTarget)
```

Description

`coder.buildstatus.open(model)` opens the **Build Process Status** window for the `model`.

The **Build Process Status** window supports parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

If the current working folder is the model build folder and the folder contains information from a previous parallel build, opening the **Build Process Status** window displays the previous build information. When you start a model parallel build, the current build information replaces the previous build information in the window.

`coder.buildstatus.open(model,systemTarget)` opens the **Build Process Status** window for the `model` and displays the `model` tab. The available tabs are **Simulation Targets** and **Code Generation Targets**

Examples

Open Build Process Status Window for a Model

After generating code for model `'rtwdemo_parabuild_a_1'`, open the **Build Process Status** window for the model.

```
addpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','rtwdemo_parallelbuild'))
coder.buildstatus.open('rtwdemo_parabuild_a_1')
```

Open Build Process Status Window with Simulation Targets

Open the **Build Process Status** window for the model `'rtwdemo_parabuild_a_1'` and display the **Simulation Targets** tab.

```
addpath(fullfile(matlabroot,'toolbox','rtw','rtwdemos','rtwdemo_parallelbuild'))
coder.buildstatus.open('rtwdemo_parabuild_a_1','sim')
```

Input Arguments

`model` — Model name

character vector | string scalar

Model name specified as a character vector or a string scalar

Example: `'rtwdemo_parabuild_a_1'`

Data Types: `char` | `string`

`systemTarget` — System targets name

`sim` | `rtw`

System targets tab name specified as a character vector or string scalar, `sim` for **Simulation Targets** and `rtw` for **Code Generation Targets**. When build information is available for a system target from a previous build, the *`systemTarget`* argument directs the **Build Status** dialog box to display the tab for the system target. If this optional argument is omitted, when build information is available, dialog opens both the **Simulation Targets** tab and **Code Generation Targets** tab. If build information for a target is not available, the dialog does not open the corresponding system targets tab.

Example: `'rtw'`

Data Types: `char` | `string`

See Also

`coder.buildstatus.close` | `coder.report.open` | `rtwbuild` | `slbuild`

Topics

“View Build Process Status”

Introduced in R2018a

coder.codedescriptor.CodeDescriptor class

Package: coder

Return information about generated code

Description

Create a `coder.codedescriptor.CodeDescriptor` object to access all the methods defined within the code descriptor API. The `coder.codedescriptor.CodeDescriptor` object describes the data interfaces, function interfaces, global data stores, local and global parameters in the generated code.

Construction

`codeDescObj = coder.getCodeDescriptor(model)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified model.

`codeDescObj = coder.getCodeDescriptor(folder)` creates a `coder.codedescriptor.CodeDescriptor` object for the model in the build folder specified in `folder`.

Properties

`ModelName` — Name of the model
character vector (default)

Name of the model for which the code descriptor object is invoked.

Example: `'rtwdemo_comments'`

`BuildFolder` — Build folder
character vector (default)

Path of the build folder where the model is built.

Example: `'C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw'`

Methods

getAllDataInterfaceTypes	Return all data interface types
getAllFunctionInterfaceTypes	Return all function interface types
getArrayLayout	Return array layout of the generated code
getDataInterfaces	Return information of the specified data interface
getDataInterfaceTypes	Return all data interface types in the generated code
getFunctionInterfaces	Return information of the specified function interface
getFunctionInterfaceTypes	Return all function interface types in the generated code
getReferencedModelCodeDescriptor	Return `coder.codedescriptor.CodeDescriptor` object for the specified referenced model
getReferencedModelNames	Return names of the referenced models

Example

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

1 Build the model.

```
rtwbuild('rtwdemo_comments')
```

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

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

The `codeDescObj` with these properties is created:

```
    ModelName: 'rtwdemo_comments'
     BuildDir: 'C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw'
```

3 Return a list of all available function interface types.

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
```

`allFunctionInterfaceTypes` has these values:

```
{'Initialize'}
{'Output'    }
{'Update'    }
{'Terminate' }
```

See Also

`getCodeDescriptor` | `coder.descriptor.DataInterface` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

getAllDataInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return all data interface types

Syntax

```
allDataInterfaceTypes = getAllDataInterfaceTypes()
```

Description

`allDataInterfaceTypes = getAllDataInterfaceTypes()` returns a list of all the data interface types available. This list is not specific to any model.

Output Arguments

`allDataInterfaceTypes` — All data interface types available
cell array of character vectors

A list of all the available data interface types.

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the required model that is built, then list all the available data interface types.

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

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

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

3 Return a list of all available data interface types.

```
allDataInterfaceTypes = getAllDataInterfaceTypes(codeDescObj)
```

`allDataInterfaceTypes` has these values:

```
    {'Inports'         }
    {'Outports'        }
    {'Parameters'      }
    {'GlobalDataStores'}
    {'GlobalParameters'}
    {'LocalParameters' }
```

In a model, there can be `GlobalParameters` and/or `LocalParameters`. The data interface type `Parameters` consist of a consolidated list of both types of parameters.

See Also

`coder.codedescriptor.CodeDescriptor` | `coder.descriptor.DataInterface` | `getDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

getAllFunctionInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return all function interface types

Syntax

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes()
```

Description

`allFunctionInterfaceTypes = getAllFunctionInterfaceTypes()` returns a list of all the function interface types available. The returned list is not specific to any model.

Output Arguments

`allFunctionInterfaceTypes` — All function interface types available
cell array of character vectors

A list of all the available function interface types.

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the required model which is built, then list all the available function interface types.

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

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

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

3 Return a list of all available function interface types.

```
allFunctionInterfaceTypes = getAllFunctionInterfaceTypes(codeDescObj)
```

`allFunctionInterfaceTypes` has these values:

```
{'Initialize'}
{'Output'    }
{'Update'    }
{'Terminate' }
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code”
“Configure Code Generation for Model Entry-Point Functions”

Introduced in R2018a

getArrayLayout

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return array layout of the generated code

Syntax

```
arrayLayout = getArrayLayout()
```

Description

`arrayLayout = getArrayLayout()` returns the array layout of the model for which the code is generated.

Output Arguments

`arrayLayout` — Array layout of the generated code
character vectors

Array layout specified for the model by using the model configuration parameter **Array layout** on page 9-71.

Examples

Create a `coder.codedescriptor.CodeDescriptor` object for the model that is built, then list the array layout of the generated code.

1. Open a model.

   ```
   rtwdemo_comments
   ```
2. Specify the model configuration parameter **Array layout** as `Row-major`. Alternatively, in the command window, use these commands:

```
set_param('rtwdemo_comments', 'ArrayLayout','Row-major');
```

3 Build the model.

```
rtwbuild('rtwdemo_comments')
```

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

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

5 Return the array layout of the generated code.

```
arrayLayout = getArrayLayout(codeDescObj)
```

`arrayLayout` has this value:

```
'Row-major'
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”
“Code Generation of Matrices and Arrays”

Introduced in R2018b

getDataInterfaces

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return information of the specified data interface

Syntax

```
dataInterface = getDataInterfaces(dataInterfaceName)
```

Description

`dataInterface = getDataInterfaces(dataInterfaceName)` returns the type of data, SID, graphical name, timing, implementation, and variant information on the data interface that `dataInterfaceName` specifies.

Input Arguments

`dataInterfaceName` — Name of data interface

`Inports` | `Outports` | `Parameters` | `GlobalDataStores` | `GlobalParameters` | `LocalParameters`

`dataInterfaceName` specifies the name of a data interface. To get a list of all the data interfaces in the generated code, call `getDataInterfaceTypes()`.

Data Types: `string`

Output Arguments

`dataInterface` — `coder.DataInterface` object with properties of specified data interface type

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

The `coder.descriptor.DataInterface` object describes information about the specified data interface such as type of data, SID, graphical name, timing, implementation, and variant information.

Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

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

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

3. Return a list of all data interface types in the generated code.

   ```
   dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
   ```

   `dataInterfaceTypes` has these values:

   ```
       {'Inports'         }
       {'Outports'        }
       {'Parameters'      }
       {'GlobalParameters'}
   ```

4. Return properties of Inport blocks in the generated code.

   ```
   dataInterface = getDataInterfaces(codeDescObj, 'Inports')
   ```

   `dataInterface` is an array of `coder.DataInterface` objects. Obtain the details of the first Inport block of the model by accessing the first location in the array.

   ```
   dataInterface(1)
   ```

   The first `coder.DataInterface` object with properties is returned.

   ```
                Type: [1×1 coder.descriptor.types.Double]
                 SID: 'rtwdemo_comments:1'
         GraphicalName: 'In1'
           VariantInfo: [0×0 coder.descriptor.VariantInfo]
       Implementation: [1×1 coder.descriptor.StructExpression]
              Timing: [1×1 coder.descriptor.TimingInterface]
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaceTypes` | `coder.descriptor.DataInterface`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

getDataInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return all data interface types in the generated code

Syntax

```
dataInterfaceTypes = getDataInterfaceTypes()
```

Description

`dataInterfaceTypes = getDataInterfaceTypes()` returns a list of all the data interface types in the generated code. To get a list of all the available data interfaces, call `getAllDataInterfaceTypes()`.

Output Arguments

`dataInterfaceTypes` — All data interface types in the generated code
cell array of character vectors

A list of all the data interface types in the generated code.

Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_counter')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
   ```
3. Return a list of all data interface types in the generated code.

   ```
   dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
   ```

`dataInterfaceTypes` has these values for model `rtwdemo_counter`:

```
{'Inports'         }
{'Outports'        }
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

getFunctionInterfaces

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return information of the specified function interface

Syntax

```
functionInterface = getFunctionInterfaces(functionInterfaceName)
```

Description

`functionInterface = getFunctionInterfaces(functionInterfaceName)` returns the function prototype, input arguments, return arguments, variant conditions, and timing information of the function interface that `functionInterfaceName` specifies.

Input Arguments

`functionInterfaceName` — Name of function interface

`Initialize` | `Output` | `Update` | `Terminate`

`functionInterfaceName` specifies the name of a function interface. A list of all the function interfaces in the generated code is returned by `getFunctionInterfaceTypes()`.

Data Types: `string`

Output Arguments

`functionInterface` — `coder.FunctionInterface` object with properties of specified function interface type

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

The `coder.descriptor.FunctionInterface` object describes information about the specified function interface such as function prototype, input arguments, return arguments, variant conditions, and timing information.

Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```

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

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

3. Return a list of all function interface types in the generated code.

   ```
   functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
   ```

   These are the function interface types in the generated code of model `rtwdemo_comments`:

   ```
       {'Initialize'}
       {'Output'    }
   ```

4. Return properties of a specified function interface in the generated code.

   ```
   functionInterface = getFunctionInterfaces(codeDescObj, 'Output')
   ```

   `functionInterface` is a `coder.FunctionInterface` object.

   ```
              Prototype: [1×1 coder.descriptor.types.Prototype]
         ActualReturn: [0×0 coder.descriptor.DataInterface]
          VariantInfo: [0×0 coder.descriptor.VariantInfo]
               Timing: [1×1 coder.descriptor.TimingInterface]
           ActualArgs: [1x0 coder.descriptor.DataInterface List]
   ```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaceTypes` | `coder.descriptor.FunctionInterface`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

getFunctionInterfaceTypes

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return all function interface types in the generated code

Syntax

```
functionInterfaceTypes = getFunctionInterfaceTypes()
```

Description

`functionInterfaceTypes = getFunctionInterfaceTypes()` returns a list of all the function interface types in the generated code. To get a list of all the available function interfaces, call `getAllFunctionInterfaceTypes()`.

Output Arguments

`functionInterfaceTypes` — All function interface types in the generated code
cell array of character vectors

A list of all the data interface types in the generated code.

Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_counter')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_counter')
   ```
3. Return a list of all function interface types in the generated code.

   ```
   functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
   ```

`functionInterfaceTypes` has these values for model `rtwdemo_counter`:

```
    {'Output'    }
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaces` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

getReferencedModelCodeDescriptor

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return `coder.codedescriptor.CodeDescriptor` object for the specified referenced model

Syntax

```
refCodeDescriptor = getReferencedModelCodeDescriptor(refModelName)
```

Description

`refCodeDescriptor = getReferencedModelCodeDescriptor(refModelName)` returns the `coder.codedescriptor.CodeDescriptor` object for the referenced model specified in `refModelName`.

Input Arguments

`refModelName` — Name of referenced model
string

`refModelName` can take any name from the list of referenced models returned by `getReferencedModelNames()`.

Output Arguments

`refCodeDescriptor` — `coder.codedescriptor.CodeDescriptor` object for the specified referenced model
`coder.codedescriptor.CodeDescriptor` object

`coder.codedescriptor.CodeDescriptor` object for the specified referenced model.

Examples

1. Build the model.

```
rtwbuild('rtwdemo_async_mdlreftop')
```

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

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

3. Return a list of referenced models.

```
refModels = getReferencedModelNames(codeDescObj)
```

`refModels` contains the list of referenced models for `rtwdemo_async_mdlreftop`.

```
{'rtwdemo_async_mdlrefbot'}
```

Obtain the `coder.codedescriptor.CodeDescriptor` object for any of the referenced models.

```
refCodeDescriptorObj = getReferencedModelCodeDescriptor(codeDescObj, 'rtwdemo_async_mdlrefbot')
```

`refCodeDescriptorObj` is the `coder.codedescriptor.CodeDescriptor` object for `rtwdemo_async_mdlrefbot` model.

```
    ModelName: 'rtwdemo_async_mdlrefbot'
     BuildDir: 'C:\Users\Desktop\Work\slprj\tornado\rtwdemo_async_mdlrefbot'
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelNames` | `getCodeDescriptor`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

getReferencedModelNames

Class: coder.codedescriptor.CodeDescriptor
Package: coder

Return names of the referenced models

Syntax

```
refModels = getReferencedModelNames()
```

Description

`refModels = getReferencedModelNames()` returns a list of referenced models for a `coder.codedescriptor.CodeDescriptor` object.

Output Arguments

`refModels` — Names of referenced models
cell array of character vectors

A list of referenced models.

Examples

1. Build the model.

   ```
   rtwbuild('rtwdemo_async_mdlreftop')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_async_mdlreftop')
   ```
3. Return a list of referenced models.

   ```
   refModels = getReferencedModelNames(codeDescObj)
   ```

`refModels` has the list of referenced models.

```
{'rtwdemo_async_mdlrefbot'}
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getReferencedModelCodeDescriptor`

Topics

"Get Code Description of Generated Code"

Introduced in R2018a

coder.descriptor.DataInterface class

Package: coder

Return information about different types of data interfaces

Description

The `coder.descriptor.DataInterface` object describes various properties for a specified data interface in the generated code. These are the different types of data interfaces:

- Root-level inports and outports: An interface between the model and external models or systems, for exchanging data.
- Block-specific parameters: Local and Global parameters that describes the data for the block.
- Global Data Store: A repository to store global data that can be written and read.

Construction

`dataInterface = getDataInterfaces(codeDescObj, dataInterfaceName)` creates a `coder.descriptor.DataInterface` object. `codeDescObj` is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

Input Arguments

`dataInterfaceName` — Name of data interface
`Inports` | `Outports` | `Parameters` | `GlobalDataStores` | `GlobalParameters` | `LocalParameters`

Name of the specified data interface.

Example: `'Inports'`

Data Types: `string`

Properties

`Type` — Type of data
`coder.descriptor.types` object

The data type associated with the data such as `integer`, `double`, `matrix`, and its properties.

`SID` — Simulink identifier
character vector

The Simulink identifier (SID) is a unique number within the model that Simulink assigns to the block.

`GraphicalName` — Name of graphical entity
character vector

The name of the associated graphical entity.

`VariantInfo` — Variant conditions in the model
`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the data interface.

`Implementation` — Description of implementation of data
`coder.descriptor.DataImplementation` object

The description of how the data in the generated code is implemented. This property describes characteristics such as data type and size. In addition, it describes how the data is accessed or declared in the code. The property describes if the data is declared as a variable or structure member.

`Timing` — Data access rate in run-time environment
`coder.descriptor.TimingInterface` object

The rate at which data is accessed in a run-time environment.

Example

1 Build the model.

```
rtwbuild('rtwdemo_comments')
```

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

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

3 Return a list of all data interface types in the generated code.

```
dataInterfaceTypes = getDataInterfaceTypes(codeDescObj)
```

`dataInterfaceTypes` has these values:

```
    {'Inports'         }
    {'Outports'        }
    {'Parameters'      }
    {'GlobalParameters'}
```

4 Return properties of a specified data interface in the generated code.

```
dataInterface = getDataInterfaces(codeDescObj, 'Inports')
```

`dataInterface` is an array of `coder.DataInterface` objects. Obtain the details of the first Inport block of the model by accessing the first location in the array.

```
dataInterface(1)
```

The first `coder.DataInterface` object with properties is returned.

```
              Type: [1×1 coder.descriptor.types.Double]
               SID: 'rtwdemo_comments:1'
     GraphicalName: 'In1'
       VariantInfo: [0×0 coder.descriptor.VariantInfo]
    Implementation: [1×1 coder.descriptor.StructExpression]
            Timing: [1×1 coder.descriptor.TimingInterface]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllDataInterfaceTypes` | `getDataInterfaceTypes` | `getDataInterfaces`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

coder.descriptor.FunctionInterface class

Package: coder

Return information about entry-point functions

Description

The function interfaces are the entry-point functions in the generated code. The `coder.descriptor.FunctionInterface` object describes various properties for a specified function interface. The different types of function interfaces are:

- Initialize: Contains initialization code for the model and is called once at the start of your application code. See `model_initialize`.
- Output: Contains the output code for the blocks in the model. See `model_step`.
- Update: Contains the update code for the blocks in the model. See `model_step`.
- Terminate: Contains the termination code for the model and is called as part of a system shutdown. See `model_terminate`.

Construction

`functionInterface = getFunctionInterfaces(codeDescObj, functionInterfaceName)` creates a `coder.descriptor.FunctionInterface` object. `codeDescObj` is the `coder.codedescriptor.CodeDescriptor` object created for the model by using the `getCodeDescriptor` function.

Input Arguments

`functionInterfaceName` — Name of function interface
`Initialize` | `Output` | `Update` | `Terminate`

Name of the specified function interface

Example: `'Output'`

Data Types: `string`

Properties

`Prototype` — Description of function prototype
`coder.descriptor.types` object

The description of the function prototype including function return value, name, arguments, header, and source files.

`ActualReturn` — Return arguments from the function
`coder.descriptor.DataInterface` object

The data that the function returns as a return argument. When there is no data returned from the function, this field is empty.

`VariantInfo` — Variant conditions in the model
`coder.descriptor.VariantInfo` object

The variant conditions in the model that interact with the function interface.

`Timing` — Function access rate in run-time environment
`coder.descriptor.TimingInterface` object

The rate at which function is accessed in a run-time environment.

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

The data passed as arguments to the function. When there is no data passed as an argument to the function, this field is empty.

Example

1. Build the model.

   ```
   rtwbuild('rtwdemo_comments')
   ```
2. Create a `coder.codedescriptor.CodeDescriptor` object for the required model.

   ```
   codeDescObj = coder.getCodeDescriptor('rtwdemo_comments')
   ```
3. Return a list of all function interface types in the generated code.

   ```
   functionInterfaceTypes = getFunctionInterfaceTypes(codeDescObj)
   ```

`functionInterfaceTypes` consists this:

```
    {'Initialize'}
    {'Output'    }
```

4 Return properties of a specified function interface in the generated code.

```
functionInterface = getFunctionInterfaces(codeDescObj, 'Output')
```

`functionInterface` is a `coder.FunctionInterface` object.

```
        Prototype: [1×1 coder.descriptor.types.Prototype]
     ActualReturn: [0×0 coder.descriptor.DataInterface]
      VariantInfo: [0×0 coder.descriptor.VariantInfo]
           Timing: [1×1 coder.descriptor.TimingInterface]
       ActualArgs: [1x0 coder.descriptor.DataInterface List]
```

See Also

`coder.codedescriptor.CodeDescriptor` | `getAllFunctionInterfaceTypes` | `getFunctionInterfaceTypes` | `getFunctionInterfaces`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

coder.report.close

Close HTML code generation report

Syntax

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

Description

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

Examples

Close code generation report for a model

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

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

See Also

`coder.report.generate` | `coder.report.open`

Topics

"Reports for Code Generation"

Introduced in R2012a

coder.report.generate

Generate HTML code generation report

Syntax

```
coder.report.generate(model)
coder.report.generate(subsystem)
coder.report.generate(model,Name,Value)
```

Description

`coder.report.generate(model)` generates a code generation report for the `model`. The build folder for the model must be present in the current working folder.

`coder.report.generate(subsystem)` generates the code generation report for the `subsystem`. The build folder for the subsystem must be present in the current working folder.

`coder.report.generate(model,Name,Value)` generates the code generation report using the current model configuration and additional options specified by one or more `Name,Value` pair arguments. Possible values for the `Name,Value` arguments are parameters on the **Code Generation > Report** pane. Without modifying the model configuration, using the `Name,Value` arguments you can generate a report with a different report configuration.

Examples

Generate Code Generation Report for Model

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

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

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

Close the code generation report.

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

Generate a code generation report.

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

Generate Code Generation Report for Subsystem

Open the model `rtwdemo_counter`.

```
open rtwdemo_counter
```

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

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

Close the code generation report.

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

Generate a code generation report for the subsystem.

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

Generate Code Generation Report to Include Static Code Metrics Report

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

Open the model `rtwdemo_hyperlinks`.

```
open rtwdemo_hyperlinks
```

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

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

Close the code generation report.

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

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

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

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

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: ‘`rtwdemo_counter`’

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: ‘`rtwdemo_counter/Amplifier`’

Data Types: `char`

Name-Value Pair Arguments

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

Each `Name,Value` argument corresponds to a parameter on the Configuration Parameters **Code Generation > Report** pane. When the configuration parameter `GenerateReport` is `on`, the parameters are enabled. The `Name,Value` arguments are

used only for generating the current report. The arguments will override, but not modify, the parameters in the model configuration. The following parameters require an Embedded Coder® license.

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

Navigation

`IncludeHyperlinkInReport` — Code-to-model hyperlinks

`'off'` | `'on'`

Code-to-model hyperlinks, specified as `'on'` or `'off'`. Specify `'on'` to include code-to-model hyperlinks in the code generation report. The hyperlinks link code to the corresponding blocks, Stateflow® objects, and MATLAB® functions in the model diagram. For more information see "Code-to-model" on page 10-6.

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 10-8.

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 5-10.

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 10-11.

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 10-13.

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 10-15.

Example: `''GenerateTraceReportSf','on''`

Data Types: `char`

`GenerateTraceReportEml` — Summary of MATLAB functions and the corresponding code location
`'off'` | `'on'`

Summary of the MATLAB functions and the corresponding code location, specified as '`on`' or '`off`'. Specify '`on`' to include a summary of the MATLAB objects and the corresponding

code location in the code generation report. For more information, see “Traceable MATLAB functions” on page 10-17.

Example: `''GenerateTraceReportEml','on''`

Data Types: `char`

Metrics

`GenerateCodeMetricsReport` — Static code metrics
`'off'` | `'on'`

Static code metrics, specified as `'on'` or `'off'`. Specify `'on'` to include static code metrics in the code generation report. For more information, see “Static code metrics” on page 5-12.

Example: `''GenerateCodeMetricsReport','on''`

Data Types: `char`

See Also

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

Topics

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

Introduced in R2012a

coder.report.open

Open existing HTML code generation report

Syntax

```
coder.report.open(model)
coder.report.open(subsystem)
```

Description

`coder.report.open(model)` opens a code generation report for the `model`. The build folder for the model must be present in the current working folder.

`coder.report.open(subsystem)` opens a code generation report for the `subsystem`. The build folder for the subsystem must be present in the current working folder.

Examples

Open code generation report for a model

After generating code for `rtwdemo_counter`, open a code generation report for the model.

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

Open code generation report for a subsystem

Open a code generation report for the subsystem ‘`Amplifier`’ in model ‘`rtwdemo_counter`’.

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

Input Arguments

`model` — Model name
character vector

Model name specified as a character vector

Example: 'rtwdemo_counter'

Data Types: `char`

`subsystem` — Subsystem name
character vector

Subsystem name specified as a character vector

Example: 'rtwdemo_counter/Amplifier'

Data Types: `char`

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

Topics
"Reports for Code Generation"
"Open Code Generation Report"

Introduced in R2012a

extmodeBackgroundRun

Perform external mode background activity

Syntax

```
errorCode = extmodeBackgroundRun();
```

Description

`errorCode = extmodeBackgroundRun();` performs external mode background activity, for example, retrieving packets from the network, running the packets protocol layer, and sending packets to the development computer.

Do not invoke the function in a thread with real-time constraints.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_BUSY` (-6) –– Resource busy detected, try later
- `EXTMODE_INV_MSG_FORMAT` (-7) –– Invalid message format detected by external mode communication protocol.
- `EXTMODE_INV_SIZE` (-8) –– Invalid size detected by the external mode communication protocol.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.
- `EXTMODE_NO_MEMORY` (-10) –– No memory available on the target hardware.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.
- `EXTMODE_PKT_CHECKSUM_ERROR` (-13) –– Checksum inconsistency detected by external mode communication protocol.
- `EXTMODE_PKT_RX_TIMEOUT_ERROR` (-14) –– Timeout error detected during the reception of a packet.
- `EXTMODE_PKT_TX_TIMEOUT_ERROR` (-15) –– Timeout error detected during the transmission of a packet.

See Also

`extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeEvent

External mode event trigger

Syntax

```
errorCode = extmodeEvent(eventId, simulationTime)
```

Description

`errorCode = extmodeEvent(eventId, simulationTime)` informs the external mode abstraction layer of the occurrence of an event.

`eventId` is the sample time ID of the model, for example, 0 for base rate, 1 for first subrate, and so on.

The function:

- Samples all signals associated with a given sample time.
- Stores signal values in a new packet buffer.
- Passes the packet buffer to the underlying transport layer for subsequent transmission to the development computer.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

For correct sampling of signal values, run the function immediately after *`model`*`_step()` for the corresponding sample time ID. You can invoke the function with different sample time IDs in separate threads because the function is thread-safe.

The `extmodeBackgroundRun` function performs the transmission of signal values to the development computer.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Input Arguments

`eventId` — Event ID
uint16_T

Sample time ID of the model, which is 0 for base rate, 1 for first subrate, 2 for second subrate, and so on.

`simulationTime` — Simulation time
real_T

Time when event occurs.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.
- `EXTMODE_NO_MEMORY` (-10) –– No memory available on the target hardware.

See Also

`extmodeBackgroundRun` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` |

`extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeGetFinalSimulationTime

Get final simulation time for external mode platform abstraction layer

Syntax

```
errorCode = extmodeGetFinalSimulationTime(finalTime);
```

Description

`errorCode = extmodeGetFinalSimulationTime(finalTime);` gets the model's final simulation time for the external mode platform abstraction layer. The function is a complementary function for `extmodeSetFinalSimulationTime`.

Output Arguments

`finalTime` — Final simulation time
real_T pointer

Final simulation time of model.

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_NOT_INITIALIZED` (-9) –– External mode not initialized yet.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` |

`extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeInit

Initialize external mode target connectivity

Syntax

```
errorCode = extmodeInit(extmodeInfo, finalTime);
```

Description

`errorCode = extmodeInit(extmodeInfo, finalTime);` initializes the external mode target connectivity, including the underlying communication stack.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Input Arguments

`extmodeInfo` — External mode information structure
RTWExtModeInfo structure

Model structure that contains information for the external mode simulation. `RTWExtModeInfo` is defined in *`matlabroot`*`/simulink/include/rtw_extmode.h`.

`finalTime` — Final simulation time
real_T pointer

If the model's final simulation time in the external mode abstraction layer is initialized, then `finalTime` is an output and the pointer location is updated with the initialized value. You might initialize the final simulation time through the `'-tf '` option detected by `extmodeParseArgs()` or `extmodeSetFinalSimulationTime()`

If the model's final simulation time in the external mode abstraction layer is not initialized, then `finalTime` is an input and the model's final simulation time in external mode is updated accordingly.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

"External Mode Simulation with XCP Communication"
"Customize XCP Slave Software"

Introduced in R2018a

extmodeParseArgs

Extract values of configuration parameters supported by external mode abstraction layer

Syntax

```
errorCode = extmodeParseArgs(argCount, argValues);
```

Description

`errorCode = extmodeParseArgs(argCount, argValues);` extracts the values of the configuration parameters that are supported by the external mode abstraction layer. The function parses the array of strings passed as input arguments. The array of strings is from the command-line arguments of the executable file running on the target hardware.

The external mode abstraction layer interprets only two options and passes the other arguments to `rtIOStreamOpen` for the initialization of the communication driver.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

If your target hardware does not support the parsing of command-line arguments, define the preprocessor macro `EXTMODE_DISABLE_ARGS_PROCESSING`. See information about parsing command-line arguments in “Other Platform Abstraction Layer Functionality”.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Input Arguments

`argCount` — Number of arguments
int_T scalar

Number of elements in `argValues` array.

`argValues` — Command-line arguments
array of null-terminated strings

Command-line arguments of the executable file running on the target hardware. The external mode abstraction layer interprets only these options:

- `'-w'` – Enables the `extmodeWaitForStartRequest()` function, which waits for a model start request from Simulink in external mode. If you do not specify this option, the `extmodeWaitForStartRequest()` function has no effect.
- `'-tf` *`finalSimulationTime`*`'` – *`finalSimulationTime`* overrides the Simulink configuration parameter, `StopTime`.

If the command contains more options, they are passed to `rtIOStreamOpen` as configuration parameters for the communication driver.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeReset

Reset external mode target connectivity

Syntax

```
errorCode = extmodeReset();
```

Description

`errorCode = extmodeReset();` restores the external mode abstraction layer, including the communication stack, to the initial, default state.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_ERROR` (-12) –– External mode generic error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeSetFinalSimulationTime

Set final simulation time in external mode platform abstraction layer

Syntax

```
errorCode = extmodeSetFinalSimulationTime(finalTime);
```

Description

`errorCode = extmodeSetFinalSimulationTime(finalTime);` sets the final simulation time of the model in the external mode platform abstraction layer.

In the `main` function of your external mode target application, before `extmodeInit`, you can call `extmodeSetFinalSimulationTime` to set the final simulation time if:

- You do not want to use `extmodeParseArgs`.
- Your target hardware does not support parsing of command-line arguments but you want to override `StopTime` from the target application.

`extmodeGetFinalSimulationTime` and `extmodeSetFinalSimulationTime` are complementary functions.

Input Arguments

`finalTime` — Final simulation time
real_T

Final simulation time of model.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSimulationComplete` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeSimulationComplete

Check that external mode simulation is complete

Syntax

```
simComplete = extmodeSimulationComplete();
```

Description

`simComplete = extmodeSimulationComplete();` during an external mode simulation, checks whether the model simulation time has reached the final simulation time specified by the command-line `'-tf'` option or the Simulink configuration parameter, `StopTime`.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Output Arguments

`simComplete` — Simulation complete
true | false

`true` if model simulation time has reached the specified final simulation time. Otherwise, returns `false`.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeStopRequested` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeStopRequested

Check whether request to stop external mode simulation is received from model

Syntax

```
stopRequest = extmodeStopRequested();
```

Description

`stopRequest = extmodeStopRequested();` checks whether a request to stop the external mode simulation is received from the Simulink model on the development computer.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Output Arguments

`stopRequest` — Stop request
true | false

`true` if request to stop external mode simulation is received. Otherwise, returns `false`.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeWaitForHostRequest`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

extmodeWaitForHostRequest

Wait for request from development computer to start or stop external mode simulation

Syntax

```
errorCode = extmodeWaitForHostRequest(timeoutInMicroseconds);
```

Description

`errorCode = extmodeWaitForHostRequest(timeoutInMicroseconds);` waits for a start or stop request from the development computer and times out when the timeout value is reached.

Use this function with other external mode functions to enable communication between Simulink and the target application during an external mode simulation. Use the function during initialization because the function is a blocking function.

Examples

Set Up External Mode Communication

For a pseudo-code example that shows how you can provide external mode communication by using the function with related functions, see “External Mode Abstraction Layer”.

Input Arguments

`timeoutInMicroseconds` — Timeout
uint32_T

Specifies the timeout value. If the value is set to `EXTMODE_WAIT_FOREVER`, the function waits indefinitely. If `'-w'` is not extracted by `extmodeParseArgs()`, the function has no effect.

Output Arguments

`errorCode` — Error detection
extmodeErrorCode_T enumeration

Error code, returned as an `extmodeErrorCode_T` enumeration with one of these values:

- `EXTMODE_SUCCESS` (0) –– No error detected.
- `EXTMODE_INV_ARG` (-1) –– Arguments invalid.
- `EXTMODE_TIMEOUT_ERROR` (-100) –– External mode timeout error detected.

See Also

`extmodeBackgroundRun` | `extmodeEvent` | `extmodeGetFinalSimulationTime` | `extmodeInit` | `extmodeParseArgs` | `extmodeReset` | `extmodeSetFinalSimulationTime` | `extmodeSimulationComplete` | `extmodeStopRequested`

Topics

“External Mode Simulation with XCP Communication”
“Customize XCP Slave Software”

Introduced in R2018a

findBuildArg

Find a specific build argument in model build information

Syntax

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

Description

`[identifier,value] = findBuildArg(buildinfo,buildArgName)` searches for a build argument from the build information.

If the build argument is present in the model build information, the function returns the name and value.

Examples

Find Build Argument in Build Information

Find a build argument and its value stored in build information `myModelBuildInfo`. Then, view the argument identifier and value.

```
load buildInfo.mat
myModelBuildInfo = buildInfo;
myBuildArgExtmodeStaticAlloc = 'EXTMODE_STATIC_ALLOC';
[buildArgId buildArgValue] = findBuildArg(buildInfo, ...
   myBuildArgExtmodeStaticAlloc);
```

```
>> buildArgId

buildArgId =

    'EXTMODE_STATIC_ALLOC'
```

```
>> buildArgValue

buildArgValue =

    '0'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`buildArgName` — Name of build argument to find in build information
character vector | string scalar

To get the build argument identifiers from the build information, use the `getBuildArgs` function.

Output Arguments

`identifier` — Name of the build argument
character vector | string scalar

`value` — Value of the build argument
character vector | string scalar

See Also

`getBuildArgs`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2014a

findIncludeFiles

Find and add include (header) files to model build information

Syntax

```
findIncludeFiles(buildinfo,extPatterns)
```

Description

`findIncludeFiles(buildinfo,extPatterns)` searches for and adds include files to the build information.

Use the `findIncludeFiles` function to:

- Search for include files in source and include paths from the build information.
- Apply the optional *`extPatterns`* argument to specify file name extension patterns for search.
- Add the found files with their full paths to the build information.
- Delete duplicate include file entries from the build information.

Examples

Find and Add Include Files to Build Information

Find include files with file name extension `.h` that are in the build information, `myModelBuildInfo`. Add the full paths for these files to the build information. View the include files from the build information.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,{fullfile(pwd,...
    'mycustomheaders')},'myheaders');
findIncludeFiles(myModelBuildInfo);
headerfiles = getIncludeFiles(myModelBuildInfo,true,false);
```

```
>> headerfiles

headerfiles =

    'W:\work\mycustomheaders\myheader.h'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`extPatterns` — Patterns of file name extensions that specify files for the search
`'*.h'` (default) | cell array of character vectors | string array

To specify files for the search, the character vectors or strings in the *`extPatterns`* argument:

- Must start with an asterisk immediately followed by a period (*.)
- Can include a combination of alphanumeric and underscore (_) characters

Example: `'*.h' '*.hpp' '*.x*'`

See Also

`addIncludeFiles` | `getIncludeFiles` | `packNGo`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006b

getBuildArgs

Get build arguments from model build information

Syntax

```
[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs,
excludeGroupIDs)
```

Description

`[identifiers,values] = getBuildArgs(buildinfo,includeGroupIDs, excludeGroupIDs)` returns build argument identifiers and values from the build information.

The function requires the *`buildinfo`*, *`identifiers`*, and *`values`* arguments. You can use optional *`includeGroupIDs`* and *`excludeGroupIDs`* arguments. These optional arguments let you include or exclude groups selectively from the build arguments returned by the function.

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

Examples

Get Build Arguments from Build Information

After you build a model, the build information is available in the `buildInfo.mat` file. This example shows how to get the build arguments from the build information object, `myModelBuildInfo`.

```
load buildInfo.mat
myModelBuildInfo = buildInfo;
[buildArgIds,buildArgValues] = getBuildArgs(myModelBuildInfo);
```

To get the value of a single build argument from the build information, use the `findBuildArg` function.

View Build Argument Identifiers

To view the build argument identifiers, display `buildArgIds`.

```
>> buildArgIds


buildArgIds =

    'GENERATE_ERT_S_FUNCTION'
    'INCLUDE_MDL_TERMINATE_FCN'
    'COMBINE_OUTPUT_UPDATE_FCNS'
    'MAT_FILE'
    'MULTI_INSTANCE_CODE'
    'INTEGER_CODE'
    'GENERATE_ASAP2'
    'EXT_MODE'
    'EXTMODE_STATIC_ALLOC'
    'EXTMODE_STATIC_ALLOC_SIZE'
    'EXTMODE_TRANSPORT'
    'TMW_EXTMODE_TESTING'
    'MODELLIB'
    'SHARED_SRC'
    'SHARED_SRC_DIR'
    'SHARED_BIN_DIR'
    'SHARED_LIB'
    'RELATIVE_PATH_TO_ANCHOR'
    'MODELREF_TARGET_TYPE'
    'ISPROTECTINGMODEL'
```

View Build Argument Values

To view the build argument values, display `buildArgValues`.

```
>> buildArgValues


buildArgValues =
```

```
'0'
'1'
'1'
'0'
'0'
'0'
'0'
'0'
'0'
'1000000'
'0'
'0'
'iirlib.lib'
''
''
''
''
''
'..'
'NONE'
'NOTPROTECTING'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`includeGroupIDs` — Group identifiers of build arguments to include in the return from the function
cell array of character vectors | string

To use the *`includeGroupIDs`* argument, view available build argument identifier groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroupIDs` — Group identifiers of build arguments to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroupIDs`* argument, view available build argument identifier groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`identifiers` — Names of the build arguments
cell array of character vectors

`values` — Values of the build arguments
cell array of character vectors

See Also

`findBuildArg`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2014a

getCodeDescriptor

Create `coder.codedescriptor.CodeDescriptor` object for model

Syntax

```
getCodeDescriptor(model)
getCodeDescriptor(folder)
```

Description

`getCodeDescriptor(model)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified model.

`getCodeDescriptor(folder)` creates a `coder.codedescriptor.CodeDescriptor` object for the specified build folder.

Examples

Create a Code Descriptor Object Using Model Name

Create a `coder.codedescriptor.CodeDescriptor` object by using model name:

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

Create a Code Descriptor Object Using Build Folder

Create a `coder.codedescriptor.CodeDescriptor` object by using build folder:

```
codeDescObj = coder.getCodeDescriptor('C:\Users\Desktop\work\rtwdemo_comments_ert_rtw')
```

Input Arguments

`model` — Name of the model
string

Model object or name for which to obtain the `coder.codedescriptor.CodeDescriptor` object. You can get the `coder.codedescriptor.CodeDescriptor` object only for the top model if the model has referenced models.

Example: `rtwdemo_comments`

Data Types: `string`

`folder` — Build folder of the model
string

Build folder of the model for which to obtain the `coder.codedescriptor.CodeDescriptor` object. You can get the `coder.codedescriptor.CodeDescriptor` object only for the top model if the model has referenced models.

Example: `C:\Users\Desktop\Work\rtwdemo_comments_ert_rtw`

Data Types: `string`

See Also

`coder.codedescriptor.CodeDescriptor`

Topics

“Get Code Description of Generated Code”

Introduced in R2018a

getCompileFlags

Get compiler options from model build information

Syntax

```
options = getCompileFlags(buildinfo,includeGroups,excludeGroups)
```

Description

`options = getCompileFlags(buildinfo,includeGroups,excludeGroups)` returns compiler options from the build information.

The function requires the *`buildinfo`* argument. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the compiler options returned by the function.

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

Examples

Get Compiler Options from Build Information

Get the compiler options stored in the build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addCompileFlags(myModelBuildInfo,{'-Zi -Wall' '-O3'}, ...
   'OPTS');
compflags = getCompileFlags(myModelBuildInfo);
```

```
>> compflags

compflags =
```

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

Get Compiler Options with Include Group Argument

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

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

```
>> compflags

compflags =

    '-Zi -Wall'
```

Get Compiler Options with Exclude Group Argument

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

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

```
>> compflags

compflags =
```

```
'-O3'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo` object

`includeGroups` — Group names of compiler options to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of compiler options to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`options` — Compiler options from the build information
cell array of character vectors

See Also

`addCompileFlags` | `getDefines` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

getDefines

Get preprocessor macro definitions from model build information

Syntax

```
[macrodefs,identifiers,values] = getDefines(buildinfo,includeGroups,
excludeGroups)
```

Description

`[macrodefs,identifiers,values] = getDefines(buildinfo,includeGroups, excludeGroups)` returns preprocessor macro definitions from the build information.

The function requires the *`buildinfo`*, *`macrodefs`*, *`identifiers`*, and *`values`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the preprocessor macro definitions returned by the function.

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

Examples

Get Macro Definitions from Build Information

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

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

```
>> defs
```

```
defs =

    '-DPROTO=first'    '-DDEBUG'    '-Dtest'    '-DPRODUCTION'

>> names

names =

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

>> values

values =

    'first'
    ''
    ''
    ''
```

Get Macro Definitions with Include Group Argument

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

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

```
>> defs

defs =

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

Get Macro Definitions with Exclude Group Argument

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

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


>> defs

defs =

    '-DPRODUCTION'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`includeGroups` — Group names of macro definitions to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of macro definitions to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`macrodefs` — Macro definitions from the build information
cell array of character vectors

The *`macrodefs`* provide the complete macro definitions with a `-D` prefix. When the function returns a definition:

- If the `-D` was not specified when the definition was added to the build information, prepends a `-D` to the definition.
- Changes a lowercase `-d` to `-D`.

`identifiers` — Names of the macros from the build information
cell array of character vectors

`values` — Values assigned to the macros from the build information
cell array of character vectors

The *`values`* provide anything specified to the right of the first equal sign in the macro definition. The default is an empty character vector (`''`).

See Also

`addDefines` | `getCompileFlags` | `getLinkFlags`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

getFullFileList

Get list of files from model build information

Syntax

```
[fPathNames,names] = getFullFileList(buildinfo,fcase)
```

Description

`[fPathNames,names] = getFullFileList(buildinfo,fcase)` returns the fully qualified paths and names of all files, or files of a selected type (source, include, or nonbuild), from the build information.

The function requires the *`buildinfo`*, *`fPathNames`*, and *`names`* arguments. You can use the optional *`fcase`* argument. This optional argument lets you include or exclude file cases selectively from file list returned by the function.

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

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. The `getFullFileList` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

Examples

Get Full File List of All Files

After building a model and loading the generated `buildInfo.mat` file, you can list the files stored in a build information variable, `myModelBuildInfo`. This example returns information for the current model and descendants (submodels).

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

Get Full File List of Source Files

If you use any of the *`fcase`* options, you limit the listing to the files stored in the `myModelBuildInfo` variable for the current model. This example returns information for the current model only (no descendants or submodels).

```
[fPathNames,names] = getFullFileList(myModelBuildInfo,'source');
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`fcase` — File case to return from the build information
`''` (default) | `'source'` | `'include'` | `'nonbuild'`

The *`fcase`* argument selects whether the function returns the full file list for all files in the build information or returns selected cases of files. If you omit the argument or specify a null character vector (`''`), the function returns all files from the build information.

Specify	Function Action
`'source'`	Returns source files from the build information.
`'include'`	Returns include files from the build information.
`'nonbuild'`	Returns nonbuild files from the build information.

Example: `'source'`

Output Arguments

`fPathNames` — Fully qualified file paths from the build information
cell array of character vectors

`names` — File names from the build information
cell array of character vectors

See Also

Topics
“Customize Post-Code-Generation Build Processing”

Introduced in R2008a

getIncludeFiles

Get include files from model build information

Syntax

```
files = getIncludeFiles(buildinfo,concatenatePaths,
replaceMatlabroot,includeGroups,excludeGroups)
```

Description

`files = getIncludeFiles(buildinfo,concatenatePaths, replaceMatlabroot,includeGroups,excludeGroups)` returns the names of include files from the build information.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the include files returned by the function.

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

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getIncludeFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getIncludeFiles`.

Examples

Get Include Paths and Files from Build Information

Get the include paths and file names from the build information, `myModelBuildInfo`.

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

```
>> incfiles

incfiles =

    [1x22 char]    [1x36 char]    [1x21 char]
```

Get Include Paths and Files with Include Group Argument

Get the names of include files in group `etc` from the build information, `myModelBuildInfo`.

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

```
>> incfiles

incfiles =

    'etc.h'     'etc_private.h'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of include paths and files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of include paths and files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: ' '

Output Arguments

`files` — Names of include files from the build information
cell array of character vectors

The names of include files that you add with the `addIncludeFiles` function. If you call the `packNGo` function, the names include files that `packNGo` found and added while packaging model code.

See Also

`addIncludeFiles` | `findIncludeFiles` | `getIncludePaths` | `getSourceFiles` | `getSourcePaths` | `updateFilePathsAndExtensions`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

getIncludePaths

Get include paths from model build information

Syntax

```
paths = getIncludePaths(buildinfo,replaceMatlabroot,includeGroups,
excludeGroups)
```

Description

`paths = getIncludePaths(buildinfo,replaceMatlabroot,includeGroups, excludeGroups)` returns the names of include file paths from the build information.

The function requires the *`buildinfo`* and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the include paths returned by the function.

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

Examples

Get Include Paths from Build Information

Get the include paths from the build information, `myModelBuildInfo`.

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

```
>> incpaths
```

```
incpaths =

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

Get Include Paths with Include Group Argument

Get the paths in group `shared` from the build information, `myModelBuildInfo`.

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

```
>> incpaths

incpaths =

    '\common\lib'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return from the function
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output that it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of include paths to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of include paths to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`paths` — Paths of include files from the build information
cell array of character vectors

See Also

`addIncludePaths` | `getIncludeFiles` | `getSourceFiles` | `getSourcePaths`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

getLinkFlags

Get link options from model build information

Syntax

```
options = getLinkFlags(buildinfo,includeGroups,excludeGroups)
```

Description

`options = getLinkFlags(buildinfo,includeGroups,excludeGroups)` returns linker options from the build information.

The function requires the *`buildinfo`* argument. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the compiler options returned by the function.

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

Examples

Get Linker Options from Build Information

Get the linker options from the build information, `myModelBuildInfo`.

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

```
>> linkflags

linkflags =
```

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

Get Linker Options with Include Group Argument

Get the linker options with the group name `Debug` from the build information, `myModelBuildInfo`.

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

```
>> linkflags

linkflags =

    '-MD -Gy'
```

Get Linker Options with Exclude Group Argument

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

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

```
>> linkflags

linkflags =
```

```
'-T'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo` object

`includeGroups` — Group names of linker options to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of linker options to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`options` — Linker options from the build information
cell array of character vectors

See Also

`addLinkFlags` | `getCompileFlags` | `getDefines`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

getNonBuildFiles

Get nonbuild-related files from model build information

Syntax

```
files = getNonBuildFiles(buildinfo, concatenatePaths,
replaceMatlabroot,includeGroups,excludeGroups)
```

Description

`files = getNonBuildFiles(buildinfo, concatenatePaths, replaceMatlabroot,includeGroups,excludeGroups)` returns the names of non-build files from the build information, such as DLL files required for a final executable or a README file.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the non-build files returned by the function.

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

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getNonBuildFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getIncludeFiles`.

Examples

Get Nonbuild Files from Build Information

Get the nonbuild file names stored in the build information, `myModelBuildInfo`.

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

```
>> nonbuildfiles

nonbuildfiles =

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

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return from function
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return from function
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output that it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of non-build files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of non-build files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`files` — Names of non-build files from the build information
cell array of character vectors

See Also

`addNonBuildFiles`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2008a

getSourceFiles

Get source files from model build information

Syntax

```
srcfiles = getSourceFiles(buildinfo,concatenatePaths,
replaceMatlabroot,includeGroups,excludeGroups)
```

Description

`srcfiles = getSourceFiles(buildinfo,concatenatePaths, replaceMatlabroot,includeGroups,excludeGroups)` returns the names of source files from the build information.

The function requires the *`buildinfo`*, *`concatenatePaths`*, and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the source files returned by the function.

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

The makefile for the model build resolves file locations based on source paths and rules. The build process does not require you to resolve the path of every file in the build information. If you specify `true` for the *`concatenatePaths`* argument, the `getSourceFiles` function returns the path for each file:

- If a path was explicitly associated with the file when it was added.
- If you called `updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getSourceFiles`.

Examples

Get Source Files from Build Information

Get the source paths and file names from the build information, `myModelBuildInfo`.

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


>> srcfiles

srcfiles =

    'test1.c'    'test2.c'    'driver.c'
```

Get Source Files with Include Group Argument

Get the names of source files in group `tests` from the build information, `myModelBuildInfo`.

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


>> incfiles

incfiles =

    'test1.c'    'test2.c'
```

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo`
object

`concatenatePaths` — Choice of whether to concatenate paths and file names in return
`false` | `true`

Specify	Function Action
`true`	Concatenates and returns each file name with its corresponding path.
`false`	Returns only file names.

Example: `true`

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of source files to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of source files to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups by using `myGroups = getGroups(buildInfo)`.

Example: ''

Output Arguments

srcfiles — Names of source files from the build information
cell array of character vectors

The names of source files that you add with the addSourceFiles function. If you call the packNGo function, the names include files that packNGo found and added while packaging model code.

See Also

addSourceFiles | getIncludeFiles | getIncludePaths | getSourcePaths | updateFilePathsAndExtensions

Topics

"Customize Post-Code-Generation Build Processing"

Introduced in R2006a

getSourcePaths

Get source paths from model build information

Syntax

```
srcpaths = getSourcePaths(buildinfo,replaceMatlabroot,includeGroups,
excludeGroups)
```

Description

`srcpaths = getSourcePaths(buildinfo,replaceMatlabroot,includeGroups, excludeGroups)` returns the names of source file paths from the build information.

The function requires the *`buildinfo`* and *`replaceMatlabroot`* arguments. You can use optional *`includeGroups`* and *`excludeGroups`* arguments. These optional arguments let you include or exclude groups selectively from the source paths returned by the function.

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

Examples

Get Source Paths from Build Information

Get the source paths from the build information, `myModelBuildInfo`.

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

```
>> srcpaths
```

```
srcpaths =

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

Get Source Paths with Include Group Argument

Get the paths in group `tests` from the build information, `myModelBuildInfo`.

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


>> srcpaths

srcpaths =

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

Get Source Paths from Build Information

Get a source path from the build information, `myModelBuildInfo`. First, get the path without replacing `$(MATLAB_ROOT)` with an absolute path. Then, get it with replacement. Here, the MATLAB root folder is `\\myserver\myworkspace\matlab`.

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


>> srcpaths{:}

ans =

$(MATLAB_ROOT)\rtw\c\src

>> srcpaths = getSourcePaths(myModelBuildInfo,true);
```

```
>> srcpaths{:}

ans =

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

Input Arguments

`buildinfo` — Name of the build information object returned by `RTW.BuildInfo` object

`replaceMatlabroot` — Choice of whether to replace the `$(MATLAB_ROOT)` token with absolute paths in return
`false` | `true`

Use the *`replaceMatlabroot`* argument to control whether the function includes the MATLAB root definition in the output it returns.

Specify	Function Action
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.
`false`	Does not replace the token `$(MATLAB_ROOT)`.

Example: `true`

`includeGroups` — Group names of source paths to include in the return from the function
cell array of character vectors | string

To use the *`includeGroups`* argument, view available groups with `myGroups = getGroups(buildInfo)`.

Example: `''`

`excludeGroups` — Group names of source paths to exclude from the return from the function
cell array of character vectors | string

To use the *`excludeGroups`* argument, view available groups with `myGroups = getGroups(buildInfo)`.

Example: `''`

Output Arguments

`srcpaths` — Paths of source files from the build information
cell array of character vectors

See Also

`addSourcePaths` | `getIncludeFiles` | `getIncludePaths` | `getSourceFiles`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

model_initialize

Initialization entry-point function in generated code for Simulink model

Syntax

```
void model_initialize(void)
```

Calling Interfaces

The calling interface generated for this function differs depending on the value of the model parameter **Code interface packaging** on page 9-29:

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

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

  For a GRT-based model, the generated *`model`*`.c` source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, you can use the **Use dynamic memory allocation for model initialization** parameter to control whether an allocation function is generated.

  - When set, you can restart code generated from the model from a single execution instance. The sequence of function calls from the `main.c` is `allocfcn`, `model_init`, `model_term`, `allocfcn`, `model_init`, `model_term`.

- When cleared,

Note If you have an Embedded Coder license, for `Nonreusable function` code interface packaging, you can use the Code Mapping Editor to customize the name of the initialize function interface. See “Override Default Naming for Individual C Entry-Point Functions” (Embedded Coder).

Description

The generated *`model`*`_initialize` function contains initialization code for a Simulink model and should be called once at the start of your application code.

Do not use the *`model`*`_initialize` function to reset the real-time model data structure (rtM).

See Also

`model_step` | `model_terminate`

Topics

“Configure Code Generation for Model Entry-Point Functions”
“Generate Code That Responds to Initialize, Reset, and Terminate Events”

Introduced before R2006a

model_step

Step routine entry point in generated code for Simulink model

Syntax

`void *model*_step(void)`

`void *model*_step*N*(void)`

Calling Interfaces

The `*model*_step` default function prototype varies depending on the **Treat each discrete rate as a separate task** (Simulink) (`EnableMultiTasking`) parameter specified for the model:

Parameter Value	Function Prototype
`Off` (single rate or multirate)	`void *model*_step(void);`
`On` (multirate)	`void *model*_step*N* (void);` (*N* is a task identifier)

The calling interface generated for this function also differs depending on the value of the model parameter **Code interface packaging** on page 9-29:

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

  For an ERT-based model, you can use the **Pass root-level I/O as** parameter to control how root-level input and output arguments are passed to the function. They can be

included in the real-time model data structure, passed as individual arguments, or passed as references to an input structure and an output structure.

Note If you have an Embedded Coder license:

- For `Nonreusable function` code interface packaging, you can use the Configure C Step Function Interface dialog box to customize a C step function interface. See "Override Default C Step Function Interface" (Embedded Coder)"Customize Generated C Function Interfaces" (Embedded Coder).
- For `C++ class` code interface packaging, you can use the **Configure C++ Class Interface** button and related controls on the **Interface** pane of the Configuration Parameters dialog box. For more information, see "Customize Generated C++ Class Interfaces" (Embedded Coder).

Description

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

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

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

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

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

`model``_step` is designed to be called at interrupt level from `rt_OneStep`, which is assumed to be invoked as a timer ISR. `rt_OneStep` calls *`model`*`_step` to execute processing for one clock period of the model. For a description of how calls to

*model*_step are generated and scheduled, see “rt_OneStep and Scheduling Considerations” (Embedded Coder).

Note If the **Single output/update function** configuration option is not selected, the software generates the following model entry point functions in place of *model*_step:

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

---

See Also

model_initialize | model_terminate

Topics

“Configure Code Generation for Model Entry-Point Functions”

Introduced before R2006a

model_terminate

Termination entry point in generated code for Simulink model

Syntax

```
void model_terminate(void)
```

Calling Interfaces

The calling interface generated for this function also differs depending on the value of the model parameter **Code interface packaging** on page 9-29:

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

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

Description

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

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

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

For an ERT-based model, the code generator produces the *model*_terminate function for a model when the **Terminate function required** configuration option is selected (the default) in the Configuration Parameters dialog box. If your application runs indefinitely, you do not need the *model*_terminate function. To suppress the function, clear the **Terminate function required** configuration option in the Configuration Parameters dialog box.

See Also

model_initialize | model_step

Topics

“Configure Code Generation for Model Entry-Point Functions”
“Generate Code That Responds to Initialize, Reset, and Terminate Events”

Introduced before R2006a

packNGo

Package generated code in zip file for relocation

Syntax

```
packNGo(buildInfo,{Name,Value})
```

Description

`packNGo(buildInfo,{Name,Value})` packages the code files in a compressed zip file so that you can relocate, unpack, and rebuild them in another development environment. The list of name-value pairs is optional.

The types of code files in the zip file include:

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)
- MAT-file that contains the build information object (`.mat` file)
- Nonbuild-related files (for example, `.dll` files and `.txt` informational files) required for a final executable
- Build-generated binary files (for example, executable `.exe` file or dynamic link library `.dll`).

  The code generator includes the build-generated binary files (if present) in the zip file. The **ignoreFileMissing** property does not apply to build-generated binary files.

Use this function to relocate files. You can then recompile the files for a specific target environment or rebuild them in a development environment in which MATLAB is not installed. By default, the function packages the files as a flat folder structure in a zip file within the code generation folder. You can customize the output by specifying name-value pairs. After relocating the zip file, use a standard zip utility to unpack the compressed file.

The `packNGo` function can potentially modify the build information passed in the first `packNGo` argument. As part of code packaging, `packNGo` can find additional files from

source and include paths recorded in the build information. When these files are found, `packNGo` adds them to the build information.

Examples

Run packNGo from Command Window

After the build process is complete, you can run packNGo from the Command Window. Use `packNGo` for zip file packaging of generated code in the file `portzingbit.zip`. Maintain the relative file hierarchy.

1. Change folders to the code generation folder. For example, using MATLAB Coder, `codegen/dll/zingbit`, or for Simulink code generation, `zingbit_grt_rtw`.
2. Load the `buildInfo` object that describes the build.
3. Run `packNGo` with property settings for `packType` and `fileName`.

```
cd codegen/dll/zingbit;
load buildInfo.mat
packNGo(buildInfo,{'packType', 'hierarchical', ...
   'fileName','portzingbit'});
```

Configure packNGo in the Simulink Editor

If you configure zip file packaging from the code generation pane, the code generator uses `packNGo` to output a zip file during the build process.

1. Select **Code Generation > Package code and artifacts**. Optionally, provide a **Zip file name**. To apply the changes, click **OK**.
2. Build the model. The code generator outputs the zip file at the end of the build process.

Configure packNGo for Simulink from the Command Line

If you configure zip file packaging with `set_param`, the code generator uses `packNGo` to output a zip file during the build process.

Use `set_param` to configure zip file packaging for model `zingbit` in the file `zingbit.zip`.

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

Input Arguments

`buildInfo` — Object that provides build information
`buildInfo` object

During the build process, the code generator places `buildInfo.mat` in the code generation folder. This MAT-file contains the `buildInfo` object. The object provides information that `packNGo` uses to produce the zip file.

Name-Value Pair Arguments

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

Example: `{'packType','flat','nestedZipFiles',true}`

`packType` — Determines whether the primary zip file contains secondary zip files or folders
`'flat'` (default) | `'hierarchical'`

If `'flat'`, package the generated code files in a zip file as a single, flat folder.

If `'hierarchical'`, package the generated code files hierarchically in a primary zip file.

Example: `{'packType','flat'}`

`nestedZipFiles` — Determines whether the paths for files in the secondary zip files are relative to the root folder of the primary zip file
`true` (default) | false

If `true`, create a primary zip file that contains three secondary zip files:

- `mlrFiles.zip` — Files in your *`matlabroot`* folder tree

- `sDirFiles.zip` — Files in and under your code generation folder
- `otherFiles.zip` — Required files not in the *`matlabroot`* or `start` folder trees

If `false`, create a primary zip file that contains folders, for example, your code generation folder and *`matlabroot`*.

Example: `{'nestedZipFiles',true}`

`fileName` — Specifies a file name for the primary zip file
`'modelOrFunctionName``.zip'` (default) | *`'myName'`*

If you do not specify the `'fileName'`-value pair, the function packages the files in a zip file named *`modelOrFunctionName`*`.zip` and places the zip file in the code generation folder.

If you specify `'fileName'` with the value, *`'myName'`*, the function creates *`myName`*`.zip` in the code generation folder.

To specify another location for the primary zip file, provide the absolute path to the location, *`fullPath/myName`*`.zip`

Example: `{'fileName','/home/user/myModel.zip'}`

`minimalHeaders` — Selects whether to include only the minimal header files
`true` (default) | `false`

If `true`, include only the minimal header files required to build the code in the zip file.

If `false`, include header files found on the include path in the zip file.

Example: `{'minimalHeaders',true}`

`includeReport` — Selects whether to include the `html` folder for your code generation report
`false` (default) | `true`

If `false`, do not include the `html` folder in the zip file.

If `true`, include the `html` folder in the zip file.

Example: `{'includeReport',false}`

`ignoreParseError` — Instruct packNGo not to terminate on parse errors
false (default) | true

If `false`, terminate on parse errors.

If `true`, do not terminate on parse errors.

Example: `{'ignoreParseError',false}`

`ignoreFileMissing` — Instruct packNGo not to terminate if files are missing
false (default) | true

If `false`, terminate on missing file errors.

If `true`, do not terminate on missing files errors.

Example: `{'ignoreFileMissing',false}`

Limitations

- The function operates on source files only, such as `*.c`, `*.cpp`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
- Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if those files are not used.

See Also

Introduced in R2006b

rsimgetrtp

Global model parameter structure

Syntax

```
parameter_structure = rsimgetrtp('model')
```

Description

`parameter_structure = rsimgetrtp('model')` forces a block update diagram action for *`model`*, a model for which you are running rapid simulations, and returns the global parameter structure for that model. The function includes tunable parameter information in the parameter structure.

The model parameter structure contains the following fields:

Field	Description
`modelChecksum`	A four-element vector that encodes the structure. The code generator uses the *`checksum`* to check whether the structure has changed since the RSim executable was generated. If you delete or add a block, and then generate a new version of the structure, the new *`checksum`* will not match the original *`checksum`*. The RSim executable detects this incompatibility in model parameter structures and exits to avoid returning incorrect simulation results. If the structure changes, you must regenerate code for the model.
`parameters`	A structure that defines model global parameters.

The `parameters` substructure includes the following fields:

Field	Description
`dataTypeName`	Name of the parameter data type, for example, `double`

Field	Description
`dataTypeID`	An internal data type identifier
`complex`	Value 1 if parameter values are complex and 0 if real
`dtTransIdx`	Internal use only
`values`	Vector of parameter values
`structParamInfo`	Information about structure and bus parameters in the model

The `structParamInfo` substructure contains these fields:

Field	Description
`Identifier`	Name of the parameter
`ModelParam`	Value 1 if parameter is a model parameter and 0 if it is a block parameter
`BlockPath`	Block path for a block parameter. This field is empty for model parameters.
`CAPIIdx`	Internal use only

It is recommended that you do not modify fields in `structParamInfo`.

The function also includes an array of substructures `map` that represents tunable parameter information with these fields:

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

Examples

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

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

param_struct =
```

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

See Also

`rsimsetrtpparam`

Topics

“Create a MAT-File That Includes a Model Parameter Structure”
“Update Diagram and Run Simulation” (Simulink)
“Default parameter behavior” on page 15-18
“Block Creation” (Simulink)
“Tune Parameters”

Introduced in R2006a

rsimsetrtpparam

Set parameters of `rtP` model parameter structure

Syntax

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

Description

`rtP = rsimsetrtpparam(rtP,idx)` expands the `rtP` structure to have `idx` sets of parameters. The `rsimsetrtpparam` utility defines the values of an existing `rtP` parameter structure. The `rtP` structure matches the format of the structure returned by `rsimgetrtp('`*`modelName`*`')`.

`rtP = rsimsetrtpparam(rtP,'paramName',paramValue)` takes an `rtP` structure with tunable parameter information and sets the values associated with `'paramName'` to be `paramValue` if possible. There can be more than one name-value pair.

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

Examples

Expand Parameter Sets

Expand the number of parameter sets in the `rtp` structure to 10.

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

Add Parameter Sets

Add three parameter sets to the parameter structure `rtp`.

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

Input Arguments

`rtP` — A parameter structure that contains the sets of parameter names and their respective values
parameter structure

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

`paramValue` — The value of the `rtP` parameter `paramName`
value of `paramName`

`paramName` — The name of the parameter set to add to the `rtP` structure
name of the parameter set

Output Arguments

`rtP` — An expanded `rtP` parameter structure that contains `idx` additional parameter sets defined by the `rsimsetrtpparam` function call
expanded `rtP` parameter structure

See Also

`rsimgetrtp`

Topics

“Create a MAT-File That Includes a Model Parameter Structure”
“Update Diagram and Run Simulation” (Simulink)
“Default parameter behavior” on page 15-18
“Block Creation” (Simulink)
“Tune Parameters”

Introduced in R2009b

rtw_precompile_libs

Rebuild precompiled libraries within model without building model

Syntax

`rtw_precompile_libs(model,build_spec)`

Description

`rtw_precompile_libs(model,build_spec)` builds libraries within *`model`*, according to the *`build_spec`* field values, and places the libraries in a precompiled folder. Model builds that use the template makefile approach support the `rtw_precompile_libs` function. Toolchain approach model builds do not support the `rtw_precompile_libs` function.

Examples

Precompile Libraries for Model

Build the libraries in *`my_model`* without building *`my_model`*.

```
% Specify the library suffix
if isunix
   suffix = '_std.a';
elseif ismac
   suffix = '_std.a';
else
   suffix = '_vcx64.lib';
end
open_system(my_model);
set_param(my_model, 'TargetLibSuffix',suffix);

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

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

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

Input Arguments

`model` — Model object or name for which to build libraries
`object` | *`'modelName'`*

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

`build_spec` — Structure with field values that provides the build specification
struct

Structure with fields that define a build specification. Fields except `rtwmakecfgDirs` are optional.

Field Values in build_spec

Specify the structure field values of the `build_spec`.

Example: `build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};`

`rtwmakecfgDirs` — Fully qualified paths to the folders containing `rtwmakecfg` files for libraries to precompile
array of paths

Uses the `Name` and `Location` elements of `makeInfo.library`, as returned by the `rtwmakecfg` function, to specify name and location of precompiled libraries. If you use the `TargetPreCompLibLocation` parameter to specify the library folder, it overrides the `makeInfo.library.Location` setting.

The specified model must contain S-function blocks that use precompiled libraries, which the `rtwmakecfg` files specify. The makefile that the build approach generates contains the library rules only if the conversion requires the libraries.

Example: `build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};`

`libSuffix` — Suffix, including the file type extension, to append to the name of each library (for example, `_std.a` or `_vcx64.lib`)
character vector

The suffix must include a period (.). Set the suffix by using either this field or the `TargetLibSuffix` parameter. If you specify a suffix with both mechanisms, the `TargetLibSuffix` setting overrides the setting of this field.

Example: `build_spec.libSuffix = '_vcx64.lib';`

`intOnlyBuild` — Selects library optimization
`'false'` (default) | `'true'`

When set to `true`, indicates that the function optimizes the libraries so that they compile from integer code only. Applies to ERT-based targets only.

Example: `build_spec.intOnlyBuild = 'false';`

`makeOpts` — Specifies an option for `rtwMake`
character vector

Specifies an option to include in the `rtwMake` command line.

Example: `build_spec.makeOpts = '';`

`addLibs` — Specifies libraries to build
cell array of structures

This cell array of structures specifies the libraries to build that an `rtwmakecfg` function does not specify. Define each structure with two fields that are character arrays:

- `libName` — Name of the library without a suffix
- `libLoc` — Location for the precompiled library

The build approach (toolchain approach or template makefile approach) lets you specify other libraries and how to build them. Use this field if you must precompile libraries.

Example: `build_spec.addLibs = 'libs_list';`

See Also

Topics

“Precompile S-Function Libraries”
“Recompile Precompiled Libraries”
“Choose Build Approach and Configure Build Process”
“Use rtwmakecfg.m API to Customize Generated Makefiles”

Introduced in R2009b

rtwbuild

Build generated code from a model

Syntax

```
rtwbuild(model)
rtwbuild(model,name,value)

rtwbuild(subsystem)

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

Description

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

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

To reduce code generation time, when rebuilding a model, `rtwbuild` provides incremental model build. The code generator rebuilds a model or submodels only when they have changed since the most recent model build. To force a top-model build, see the `'ForceTopModelBuild'` argument.

Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (Parallel Computing Toolbox) (for example, within a `parfor` or `spmd` loop). For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models”.

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

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

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

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

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

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

Examples

Generate Code and Build Executable Image for Model

Generate C code for model `rtwdemo_rtwintro`.

```
rtwbuild('rtwdemo_rtwintro')
```

For the GRT system target file, the code generator produces the following code files and places them in folders `rtwdemo_rtwintro_grt_rtw` and `slprj/grt/_sharedutils`.

Model Files	Shared Files	Interface Files	Other Files

`rtwdemo_rtwintro.c`
`rtwdemo_rtwintro.h`
`rtwdemo_rtwintro_private.h`
`rtwdemo_rtwintrotypes.h`
	`rtGetInf.c`
	`rtGetInf.h`
	`rtGetNaN.c`
	`rtGetNaN.h`
	`rt_nonfinite.c`
	`rt_nonfinite.h`
	`rtwtypes.h`
	`multiword_types.h`
	`builtin_typeid_types.h`
		`rtmodel.h`	`rt_logging.c`

If the following model configuration parameters settings apply, the code generator produces additional results.

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

Force Top Model Build

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

```
rtwbuild('rtwdemo_mdlreftop', ...
    'ForceTopModelBuild',true)
```

Display Error Messages in Diagnostic Viewer

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

```
rtwbuild('rtwdemo_mdlreftop_witherr', ...
    'OkayToPushNags',true)
```

Generate Code and Build Executable Image for Subsystem

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

```
rtwbuild('rtwdemo_rtwintro/Amplifier')
```

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

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

If you apply the parameter settings listed in the table, the code generator produces the results listed.

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

Build Subsystem for Exporting Code to External Application

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

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

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

Create SIL Block for Verification

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

Open subsystem `rtwdemo_subsystem` in model `rtwdemo_exporting_functions` and set **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** to `SIL`.

Create the SIL block.

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

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

Name Exported Initialization Function

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

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

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

Display Status Information in Build Process Status Window

Display build information in the Build Process Status Window while generating code and running a parallel build of model `rtwdemo_mdlreftop_witherr`.

```
rtwbuild('rtwdemo_mdlreftop_witherr', ...
    'OpenBuildStatusAutomatically',true)
```

Input Arguments

`model` — Model object or name for which to generate code or build an executable image
`object` | *`'modelName'`*

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

Example: `'rtwdemo_exporting_functions'`

`subsystem` — Subsystem name for which to generate code or build executable image
`'subsystemName'`

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

Example: `'rtwdemo_exporting_functions/rtwdemo_subsystem'`

`name,value` — Name-value pairs select options for the build process
name-value pairs

Name-Value Pair Arguments

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

Example: `rtwbuild('rtwdemo_mdlreftop','ForceTopModelBuild',true)`

`ForceTopModelBuild` — Force regeneration of top model code
`false` (default) | `true`

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

Action	Specify
Force the code generator to regenerate code for the top model of a system that includes referenced models	`true`
Specify that the code generator determine whether to regenerate top model code based on model and model parameter changes	`false`

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

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

Alternatively, you can force regeneration of top model code by deleting folders in the code generation folder (Simulink), such as `slprj` or the generated model code folder.

`OkayToPushNags` — Display build error messages in Diagnostic Viewer
`false` (default) | `true`

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

Action	Specify
Display build error messages in the Diagnostic Viewer and in the Command Window	`true`

Action	Specify
Display build error messages in the Command Window only	`false`

`generateCodeOnly` — Specify code generation versus an executable build
`false` (default) | `true`

Specify code generation versus an executable build, specified as `true` or `false`.

Action	Specify
Specify code generation (same operation as value `'on'` for `GenCodeOnly` parameter)	`true`
Specify executable build (same operation as value `'off'` for `GenCodeOnly` parameter)	`false`

`Mode` — (for subsystem builds only) Direct code generator to export function calls
`'ExportFunctionCalls'` (default)

If you have Embedded Coder, generates code from `subsystem` that includes function calls that you can export to external application code.

`OpenBuildStatusAutomatically` — Display build information in the Build Process Status Window
`false` (default) | `true`

Display build information in the **Build Process Status** window, specified as `true` or `false`. For more information about using the status window, see “View Build Process Status”.

The **Build Process Status** window support parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (non-parallel) builds.

Action	Specify
Display build information in the Build Process Status Window	`true`
No action	`false`

`ObfuscateCode` — Generate obfuscated C code
false (default) | true

Specify whether to generate obfuscated C code, specified as `true` or `false`.

Action	Specify
Generate obfuscated C code that you can share with third parties with reduced likelihood of compromising intellectual property.	`true`
No action.	`false`

Output Arguments

`blockHandle` — Handle to SIL block created for generated subsystem code
handle

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

- You are licensed to use Embedded Coder software.
- **Configuration Parameters** > **Code Generation** > **Verification** > **Advanced parameters** > **Create block** is set to `SIL`.

Tips

You can initiate code generation and the build process by:

- Pressing **Ctrl+B**.
- Selecting **Code** > **C/C++ Code** > **Build Model**.
- Invoking the `slbuild` command from the MATLAB command line.

See Also

`coder.buildstatus.close` | `coder.buildstatus.open` | `rtwrebuild` | `slbuild`

Topics

“Build and Run a Program”
“Choose Build Approach and Configure Build Process”
“Control Regeneration of Top Model Code”

“Generate Component Source Code for Export to External Code Base” (Embedded Coder)
“Software-in-the-Loop Simulation” (Embedded Coder)

Introduced in R2009a

RTW.getBuildDir

Get build folder information from model build information

Syntax

```
RTW.getBuildDir(model)
folderStruct = RTW.getBuildDir(model)
```

Description

`RTW.getBuildDir(model)` displays build folder information for model.

If the model is closed, the function opens and then closes the model, leaving it in its original state. If the model is large and closed, the `RTW.getBuildDir` function can take longer to execute.

`folderStruct = RTW.getBuildDir(model)` returns a structure containing build folder information.

You can use this function in automated scripts to determine the build folder in which the generated code for a model is placed.

This function can return build folder information for protected models.

Examples

Display Build Folder Information

Display build folder information for the model `'sldemo_fuelsys'`.

```
>> RTW.getBuildDir('sldemo_fuelsys')


ans =
```

```
              BuildDirectory: 'C:\work\modelref\sldemo_fuelsys_ert_rtw'
                 CacheFolder: 'C:\work\modelref'
               CodeGenFolder: 'C:\work\modelref'
            RelativeBuildDir: 'sldemo_fuelsys_ert_rtw'
              BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\sldemo_fuelsys'
      ModelRefRelativeSimDir: 'slprj\sim\sldemo_fuelsys'
      ModelRefRelativeHdlDir: 'slprj\hdl\sldemo_fuelsys'
           ModelRefDirSuffix: ''
           SharedUtilsSimDir: 'slprj\sim\_sharedutils'
           SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

Get Build Folder Information

Return a structure `my_folderStruct` that contains build folder information for the model `'MyModel'`.

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


my_folderStruct =

              BuildDirectory: 'H:\MyModel_ert_rtw'
                 CacheFolder: 'H:\'
               CodeGenFolder: 'H:\'
            RelativeBuildDir: 'MyModel_ert_rtw'
              BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\MyModel'
      ModelRefRelativeSimDir: 'slprj\sim\MyModel'
      ModelRefRelativeHdlDir: 'slprj\hdl\MyModel'
           ModelRefDirSuffix: ''
```

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

Input Arguments

`model` — Model object or name for which to get the build folders

`object` | *`'modelName'`*

Model for which to get the build folder, specified as an object or a character vector representing the model name.

Example: `'sldemo_fuelsys'`

Output Arguments

`folderStruct` — Structure with field values that provide build folder information

struct

Structure with fields that provides build folder information.

Example: folderstruct = RTW.getBuildDir('MyModel')

`BuildDirectory` — Character vector specifying fully qualified path to build folder for model

character vector

`CacheFolder` — Character vector specifying root folder in which to place model build artifacts used for simulation

character vector

`CodeGenFolder` — Character vector specifying root folder in which to place code generation files

character vector

`RelativeBuildDir` — Character vector specifying build folder relative to the current working folder (`pwd`)

character vector

`BuildDirSuffix` — Character vector specifying suffix appended to model name to create build folder
character vector

`ModelRefRelativeRootSimDir` — Character vector specifying the relative root folder for the model reference target simulation folder
character vector

`ModelRefRelativeRootTgtDir` — Character vector specifying the relative root folder for the model reference target build folder
character vector

`ModelRefRelativeBuildDir` — Character vector specifying model reference target build folder relative to current working folder (`pwd`)
character vector

`ModelRefRelativeSimDir` — Character vector specifying model reference target simulation folder relative to current working folder (`pwd`)
character vector

`ModelRefRelativeHdlDir` — Character vector specifying model reference target HDL folder relative to current working folder (`pwd`)
character vector

`ModelRefDirSuffix` — Character vector specifying suffix appended to system target file name to create model reference build folder
character vector

`SharedUtilsSimDir` — Character vector specifying the shared utility folder for simulation
character vector

`SharedUtilsTgtDir` — Character vector specifying the shared utility folder for code generation
character vector

See Also

`rtwbuild`

Topics

“Working Folder”
“Manage Build Process Folders”

Introduced in R2008b

rtwrebuild

Rebuild generated code from model

Syntax

```
rtwrebuild()

rtwrebuild(model)

rtwrebuild(path)
```

Description

`rtwrebuild()` assumes that the current working folder is the build folder of the model (not the model location) and invokes the makefile in the build folder. If the current working folder is not the build folder, the function exits with an error.

`rtwrebuild` invokes the makefile generated during the previous build to recompile files you modified since that build. Operation of this function depends on the current working folder, not the current loaded model. If your model includes referenced models, `rtwrebuild` invokes the makefile for referenced model code recursively before recompiling the top model.

Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (Parallel Computing Toolbox) (for example, within a `parfor` or `spmd` loop). For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models”.

`rtwrebuild(model)` assumes that the current working folder is one level above the build folder and invokes the makefile in the build folder. If the current working folder (`pwd`) is not one level above the build folder, the function exits with an error.

`rtwrebuild(path)` finds the build folder indicated with the *`path`* argument and invokes the makefile in the build folder. The *`path`* argument syntax lets the function operate without regard to the relationship between the current working folder and the build folder of the model.

Examples

Rebuild Code from Build Folder

Invoke the makefile and recompile code when the current working folder is the build folder. For example,

- If the model name is `mymodel`
- And, if the model build was initiated in the `C:\work` folder
- And, if the system target is GRT

Invoke the previously generated makefile in the current working folder (build folder) `C:\work\mymodel_grt_rtw`.

```
rtwrebuild()
```

Rebuild Code from Parent Folder of Build Folder

When the current working folder is one level above the build folder, invoke the makefile and recompile code.

```
rtwrebuild('mymodel')
```

Rebuild Code from Any Folder

Invoke the makefile and recompile code from any current folder by specifying a path to the model build folder, `C:\work\mymodel_grt_rtw`.

```
rtwrebuild(fullfile('C:','work','mymodel_grt_rtw'))
```

Input Arguments

`model` — Model object or name for which to regenerate code or rebuild an executable image

`object` | `'modelName'`

Model for which to regenerate code or rebuild an executable image, specified as an object or a character vector representing the model name.

Example: `'rtwdemo_exporting_functions'`

path — Model path object or fully qualified path to the build folder for the model for which to regenerate code or rebuild an executable image
`object` | *`modelPath`*

Example: `fullfile('C:','work','mymodel_grt_rtw')`

See Also

`rtwbuild` | `slbuild`

Topics

“Rebuild a Model”

Introduced in R2009a

rtwreport

Create generated code report for model with Simulink Report Generator

Syntax

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

Description

`rtwreport(model)` creates a report of code generation information for a model. Before creating the report, the function loads the model and generates code. The code generator names the report `codegen.html`. It places the file in your current folder. The report includes:

- Snapshots of the model, including subsystems.
- Block execution order list.
- Code generation summary with a list of generated code files, configuration settings, a subsystem map, and a traceability report.
- Full listings of generated code that reside in the build folder.

`rtwreport(model,folder)` specifies the build folder, *`model_target`*`_rtw`. The build folder (`folder`) and `slprj` folder must reside in the code generation folder (Simulink). If the software cannot find the `folder`, an error occurs and code is not generated.

Examples

Create Report Documenting Generated Code

Create a report for model `rtwdemo_secondOrderSystem`:

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

Create Report Specifying Build Folder

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

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

Input Arguments

`model` — Model name
character vector

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

Example: `'rtwdemo_secondOrderSystem'`

Data Types: `char`

`folder` — Build folder name
character vector

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

Example: `'rtwdemo_secondOrderSystem_grt_rtw'`

Data Types: `char`

See Also

Topics

“Document Generated Code with Simulink Report Generator”
`Import Generated Code`
“Working with the Report Explorer” (Simulink Report Generator)
`Code Generation Summary`

Introduced in R2007a

rtwtrace

Trace a block to generated code in HTML code generation report

Syntax

```
rtwtrace('blockpath')
rtwtrace('Simulink_identifier')
rtwtrace('blockpath', 'hdl')
rtwtrace('blockpath', 'plc')
```

Description

`rtwtrace('blockpath')` opens an HTML code generation report that displays contents of the source code file and highlights the line of code corresponding to the specified block.

Before calling `rtwtrace`, make sure that:

- You select an ERT-based model and enable model to code navigation.

  In the Configuration Parameters dialog box, select the **Model-to-code** on page 10-8 parameter.
- You generate code for the model by using the code generator.
- Your build folder is under the current working folder. Otherwise, `rtwtrace` might produce an error.

`rtwtrace('Simulink_identifier')` opens an HTML code generation report that displays contents of the source code file and highlights the line of code corresponding to the block identified by the Simulink identifier (SID). SID is a unique designation for each block or element in the model. For more information, see “Locate Diagram Components Using Simulink Identifiers” (Simulink).

`rtwtrace('blockpath', 'hdl')` opens an HTML code generation report in HDL Coder™ that displays contents of the source code file and highlights the line of code corresponding to the specified block.

`rtwtrace('blockpath', 'plc')` opens an HTML code generation report in Simulink PLC Coder™ that displays contents of the source code file and highlights the line of code corresponding to the specified block.

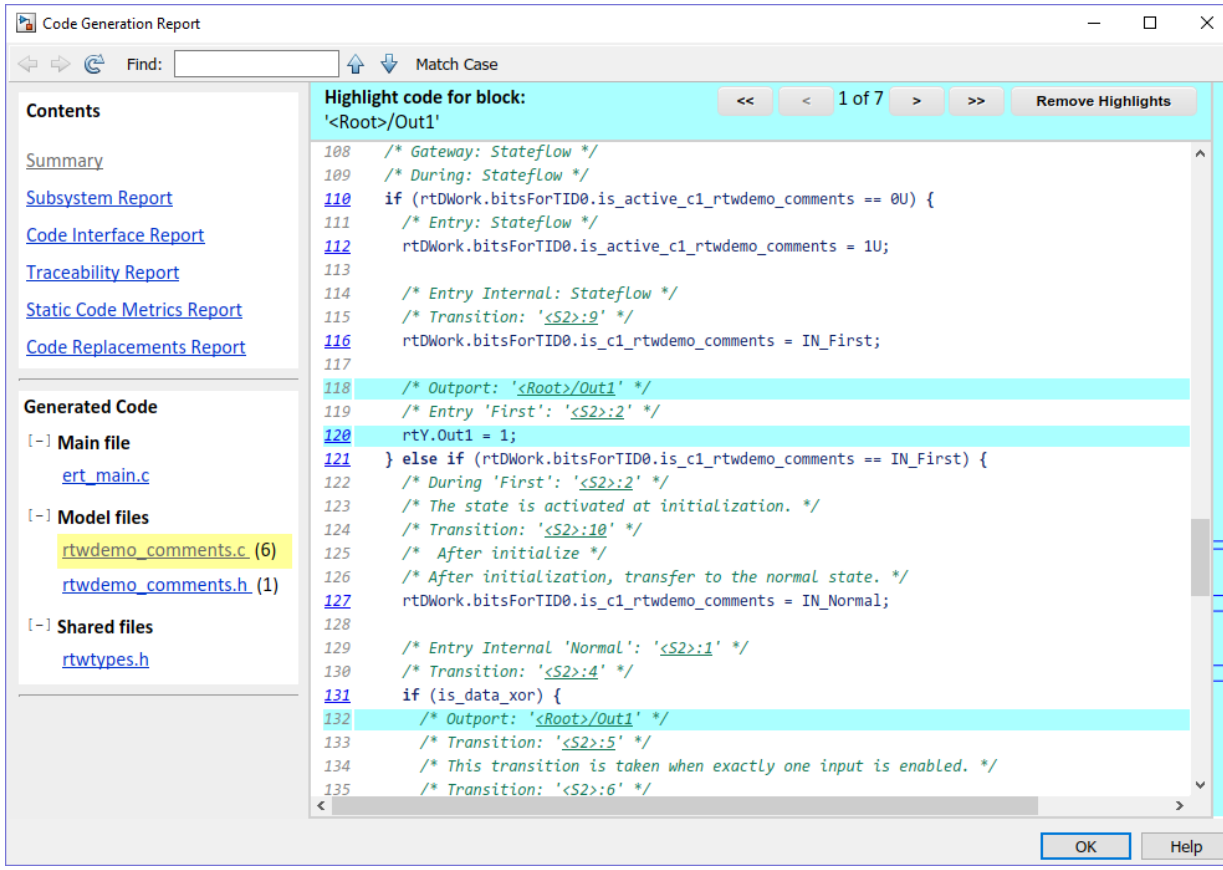
Examples

Display Generated Code for a Block

Display the generated code for block `Out1` in the model `rtwdemo_comments` in HTML code generation report:

```
% Using block path
rtwtrace('rtwdemo_comments/Out1')

% Using Simulink identifier
rtwtrace('rtwdemo_comments:33')
```



Input Arguments

`blockpath` — block path

character vector (default)

`blockpath` is a character vector enclosed in quotes specifying the full Simulink block path, for example, '*`model_name/block_name`*'.

Example: `'rtwdemo_comments/Out1'`

Data Types: `char`

`Simulink_identifier` — Simulink identifier
character vector (default)

`Simulink_identifier` is a character vector enclosed in quotes specifying the Simulink identifier, for example, `'model_name:number'`.

Example: `'rtwdemo_comments:33'`

Data Types: `char`

`hdl` — HDL Coder
character vector

`hdl` is a character vector enclosed in quotes specifying that the code report is from HDL Coder.

Example: `'Out1'`

Data Types: `char`

`plc` — PLC Coder
character vector

`plc` is a character vector enclosed in quotes specifying that the code report is from Simulink PLC Coder.

Example: `'Out1'`

Data Types: `char`

Alternatives

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

See Also

Topics

“Model-to-Code Traceability” (Embedded Coder)
“Model-to-code” on page 10-8

Introduced in R2009b

setTargetProvidesMain

Disable inclusion of code generator provided (generated or static) `main.c` source file during model build

Syntax

```
setTargetProvidesMain(buildinfo,providesmain)
```

Description

`setTargetProvidesMain(buildinfo,providesmain)` disables the code generator from including a sample `main.c` source file.

To replace the sample `main.c` file from the code generator with a custom `main.c` file, call the `setTargetProvidesMain` function during the `'after_tlc'` case in the `ert_make_rtw_hook.m` or `grt_make_rtw_hook.m` file.

Examples

Workflow for `setTargetProvidesMain`

To apply the `setTargetProvidesMain` function:

Add `buildInfo` to the arguments in the function call.

```
function ert_make_rtw_hook(hookMethod,modelName,rtwroot, ...
   templateMakefile,buildOpts,buildArgs,buildInfo)
```

Add the `setTargetProvidesMain` function to the `'after_tlc'` stage.

```
case 'after_tlc'
 % Called just after to invoking TLC Compiler (actual code generation.)
 % Valid arguments at this stage are hookMethod, modelName, and
 % buildArgs, buildInfo
```

```
%
setTargetProvidesMain(buildInfo,true);
```

Use the **Configuration Parameters > Code Generation > Custom Code > Source Files** field to add your custom `main.c` to the model. When you indicate that the target provides `main.c`, the model requires this file to build without errors.

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`providesmain` — Logical value that specifies whether the code generator includes the target provided `main.c` file
false (default) | true

The *`providesmain`* argument specifies whether the code generator includes a (generated or static) `main.c` source file.

- `false` — The code generator includes a sample `main.obj` object file.
- `true` — The target provides the `main.c` source file.

See Also
`addSourceFiles` | `addSourcePaths`

Topics
“Customize Build Process with STF_make_rtw_hook File”

Introduced in R2009a

Simulink.fileGenControl

Specify root folders for files generated by diagram updates and model builds

Syntax

```
cfg = Simulink.fileGenControl('getConfig')
Simulink.fileGenControl(Action,Name,Value)
```

Description

`cfg = Simulink.fileGenControl('getConfig')` returns a handle to an instance of the `Simulink.FileGenConfig` object, which contains the current values of these file generation control parameters:

- `CacheFolder` – Specifies the root folder for model build artifacts that are used for simulation, including Simulink® cache files.
- `CodeGenFolder` – Specifies the root folder for code generation files.
- `CodeGenFolderStructure` – Controls the folder structure within the code generation folder.

To get or set the parameter values, use the `Simulink.FileGenConfig` object.

These Simulink preferences determine the initial parameter values for the MATLAB session:

- Simulation cache folder (Simulink) – `CacheFolder`
- Code generation folder (Simulink) – `CodeGenFolder`
- Code generation folder structure (Simulink) – `CodeGenFolderStructure`

`Simulink.fileGenControl(Action,Name,Value)` performs an action that uses the file generation control parameters of the current MATLAB session. Specify additional options with one or more `name,value` pair arguments.

Examples

Get File Generation Control Parameter Values

To obtain the file generation control parameter values for the current MATLAB session, use `getConfig`.

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

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

Set File Generation Control Parameters by Using `Simulink.FileGenConfig` Object

To set the file generation control parameter values for the current MATLAB session, use the `setConfig` action. First, set values in an instance of the `Simulink.FileGenConfig` object. Then, pass the object instance. This example assumes that your system has `aNonDefaultCacheFolder` and `aNonDefaultCodeGenFolder` folders.

```
% Get the current configuration
cfg = Simulink.fileGenControl('getConfig');

% Change the parameters to non-default locations
% for the cache and code generation folders
cfg.CacheFolder = fullfile('C:','aNonDefaultCacheFolder');
cfg.CodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');
cfg.CodeGenFolderStructure = 'TargetEnvironmentSubfolder';

Simulink.fileGenControl('setConfig', 'config', cfg);
```

Set File Generation Control Parameters Directly

You can set file generation control parameter values for the current MATLAB session without creating an instance of the `Simulink.FileGenConfig` object. This example assumes that your system has `aNonDefaultCacheFolder` and `aNonDefaultCodeGenFolder` folders.

```
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ...
   'CodeGenFolder', myCodeGenFolder, ...
   'CodeGenFolderStructure', ...
    Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);
```

If you do not want to generate code for different target environments in separate folders, for `'CodeGenFolderStructure'`, specify the value `Simulink.filegen.CodeGenFolderStructure.ModelSpecific`.

Reset File Generation Control Parameters

You can reset the file generation control parameters to values from Simulink preferences.

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

Create Simulation Cache and Code Generation Folders

To create file generation folders, use the `set` action with the `'createDir'` option. You can keep previous file generation folders on the MATLAB path through the `'keepPreviousPath'` option.

```
%
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', ...
   'CacheFolder',myCacheFolder, ...
   'CodeGenFolder',myCodeGenFolder, ...
   'keepPreviousPath',true, ...
   'createDir',true);
```

Input Arguments

`Action` — Specify action

`'reset'` | `'set'` | `'setConfig'`

Specify an action that uses the file generation control parameters of the current MATLAB session:

- `'reset'` – Reset file generation control parameters to values from Simulink preferences.
- `'set'` – Set file generation control parameters for the current MATLAB session by directly passing values.
- `'setConfig'` – Set file generation control parameters for the current MATLAB session by using an instance of a `Simulink.FileGenConfig` object.

Name-Value Pair Arguments

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

Example: `Simulink.fileGenControl(`*`Action`*`, `*`Name`*`, `*`Value`*`);`

`config` — Specify instance of `Simulink.FileGenConfig`

object handle

Specify the `Simulink.FileGenConfig` object instance containing file generation control parameters that you want to set.

Option for `setConfig`.

Example: `Simulink.fileGenControl('setConfig', 'config', `*`cfg`*`);`

`CacheFolder` — Specify simulation cache folder

character vector

Specify a simulation cache folder path value for the `CacheFolder` parameter.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', `*`myCacheFolder`*`);`

`CodeGenFolder` — Specify code generation folder

character vector

Specify a code generation folder path value for the `CodeGenFolder` parameter. You can specify an absolute path or a path relative to build folders. For example:

- `'C:\Work\mymodelsimcache'` and `'/mywork/mymodelgencode'` specify absolute paths.
- `'mymodelsimcache'` is a path relative to the current working folder (`pwd`). The software converts a relative path to a fully qualified path at the time the `CacheFolder` or `CodeGenFolder` parameter is set. For example, if `pwd` is `'/mywork'`, the result is `'/mywork/mymodelsimcache'`.
- `'../test/mymodelgencode'` is a path relative to `pwd`. If `pwd` is `'/mywork'`, the result is `'/test/mymodelgencode'`.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CodeGenFolder', myCodeGenFolder);`

`CodeGenFolderStructure` — Specify generated code folder structure
`Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) | `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder`

Specify the layout of subfolders within the generated code folder:

- `Simulink.filegen.CodeGenFolderStructure.ModelSpecific` (default) – Place generated code in subfolders within a model-specific folder.
- `Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder` – If models are configured for different target environments, place generated code for each model in a separate subfolder. The name of the subfolder corresponds to the target environment.

Option for `set`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ... 'CodeGenFolder', myCodeGenFolder, ... 'CodeGenFolderStructure', ... Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);`

`keepPreviousPath` — Keep previous folder paths on MATLAB path
false (default) | true

Specify whether to keep the previous values of `CacheFolder` and `CodeGenFolder` on the MATLAB path:

- `true` – Keep previous folder path values on MATLAB path.

- `false` (default) – Remove previous older path values from MATLAB path.

Option for `reset`, `set`, or `setConfig`.

Example: `Simulink.fileGenControl('reset', 'keepPreviousPath', true);`

`createDir` — Create folders for file generation
false (default) | true

Specify whether to create folders for file generation if the folders do not exist:

- `true` – Create folders for file generation.
- `false` (default) – Do not create folders for file generation.

Option for `set` or `setConfig`.

Example: `Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, 'CodeGenFolder', myCodeGenFolder, 'keepPreviousPath', true, 'createDir',true);`

Avoid Naming Conflicts

Using `Simulink.fileGenControl` to set `CacheFolder` and `CodeGenFolder` adds the specified folders to your MATLAB search path. This function has the same potential for introducing a naming conflict as using `addpath` to add folders to the search path. For example, a naming conflict occurs if the folder that you specify for `CacheFolder` or `CodeGenFolder` contains a model file with the same name as an open model. For more information, see “What Is the MATLAB Search Path?” (MATLAB) and “Files and Folders that MATLAB Accesses” (MATLAB).

To use a nondefault location for the simulation cache folder or code generation folder:

1. Delete any potentially conflicting artifacts that exist in:
   - The current working folder, `pwd`.
   - The nondefault simulation cache and code generation folders that you intend to use.
2. Specify the nondefault locations for the simulation cache and code generation folders by using `Simulink.fileGenControl` or Simulink preferences.

Output Arguments

`cfg` — Current values of file generation control parameters
object handle

Instance of a `Simulink.FileGenConfig` object, which contains the current values of file generation control parameters.

See Also

“Simulation cache folder” (Simulink) | “Code generation folder” (Simulink) | Code generation folder structure (Simulink)

Topics

“Manage Build Process Folders”
“Share Simulation Builds for Faster Simulations” (Simulink)

Introduced in R2010b

Simulink.ModelReference.modifyProtectedModel

Modify existing protected model

Syntax

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

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

Description

`Simulink.ModelReference.modifyProtectedModel(model)` modifies options for an existing protected model created from the specified `model`. If `Name,Value` pair arguments are not specified, the modified protected model is updated with default values and supports only simulation.

`Simulink.ModelReference.modifyProtectedModel(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. These options are the same options that are provided by the `Simulink.ModelReference.protect` function. However, these options have additional options to change encryption passwords for read-only view, simulation, and code generation. When you add functionality to the protected model or change encryption passwords, the unprotected model must be available. The software searches for the model on the MATLAB path. If the model is not found, the software reports an error.

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

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

Examples

Update Protected Model with Default Values

Create a modifiable protected model with support for code generation, then reset it to default values.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Modify the model to use default values.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter');
```

The resulting protected model is updated with default values and supports only simulation.

Remove Functionality from Protected Model

Create a modifiable protected model with support for code generation and Web view, then modify it to remove the Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Webview',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Remove support for Web view from the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter', 'Mode', 'CodeGeneration','Report',true);
```

Change Encryption Password for Code Generation

Change an encryption password for a modifiable protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add the password that the protected model user must provide to generate code.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','cgpassword');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Encrypt',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Change the encryption password for simulation.

```
Simulink.ModelReference.modifyProtectedModel(
'sldemo_mdlref_counter','Mode','CodeGeneration','Encrypt',true,...
'Report',true,'ChangeSimulationPassword',...
{'cgpassword','new_password'});
```

Add Harness Model for Protected Model

Add a harness model for an existing protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add a harness model for the protected model.

```
[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Report',true,...
'Harness',true);
```

Input Arguments

`model` — Model name
string or character vector (default)

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

Name-Value Pair Arguments

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

Example: `'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

General

`Path` — Folder for protected model
current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.

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

`Report` — Option to generate a report
false (default) | true

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

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: `'Report',true`

`Harness` — Option to create a harness model
false (default) | true

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

Example: `'Harness',true`

`CustomPostProcessingHook` — Option to add postprocessing function for protected model files
function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. The object also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

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

Functionality

`Mode` — Model protection mode
`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'ViewOnly'`

Model protection mode. Specify one of the following values:

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

Example: `'Mode','Accelerator'`

`OutputFormat` — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

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

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

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

Example: `'OutputFormat','AllReferencedHeaders'`

`ObfuscateCode` — Option to obfuscate generated code

true (default) | false

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation is enabled for the protected model.

Example: `'ObfuscateCode',true`

`Webview` — Option to include a Web view

false (default) | true

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

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdlref_counter`, you can call:

  ```
  Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');
  ```

- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

Encryption

`ChangeSimulationPassword` — Option to change the encryption password for simulation

cell array of two character vectors

Option to change the encryption password for simulation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeSimulationPassword',{'old_password','new_password'}`

`ChangeViewPassword` — Option to change the encryption password for read-only view

cell array of two character vectors

Option to change the encryption password for read-only view, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeViewPassword',{'old_password','new_password'}`

`ChangeCodeGenerationPassword` — Option to change the encryption password for code generation

cell array of two character vectors

Option to change the encryption password for code generation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example:
```
'ChangeCodeGenerationPassword',
{'old_password','new_password'}
```

`Encrypt` — Option to encrypt protected model

false (default) | true

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

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

- Password for code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`

Example: `'Encrypt',true`

Output Arguments

`harnessHandle` — Handle of the harness model
double

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

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

`neededVars` — Names of base workspace variables
cell array

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

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

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForModify` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.protect

Obscure referenced model contents to hide intellectual property

Syntax

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

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

Description

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

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

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

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

Examples

Protect Referenced Model

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

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

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

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

Place Protected Model in Specified Folder

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

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

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

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

Generate Code for Protected Model

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

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

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

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

Control Code Visibility for Protected Model

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

```
sldemo_mdlref_bus;
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Mode','CodeGeneration','OutputFormat',...
'CompiledBinaries');
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the current working folder. Users can view only binary files and headers in the code generated for the protected model.

Create Harness Model for Protected Model

Create a harness model for a protected model and generate an HTML report.

```
sldemo_mdlref_bus;
modelPath= 'sldemo_mdlref_bus/CounterA'

[harnessHandle] = Simulink.ModelReference.protect(modelPath,'Path','C:\Work',...
'Harness',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created, along with an untitled harness model. The protected model file is placed in the `C:\Work` folder. The folder also contains an HTML report. The handle of the harness model is returned in `harnessHandle`.

Input Arguments

`model` — Model name
string or character vector (default)

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

Name-Value Pair Arguments

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

Example:
`'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',tr`

ue 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. Specify one of the following values:

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

Example: `'Mode','Accelerator'`

`CodeInterface` — Interface through which generated code is accessed by Model block
`'Model reference'` (default) | `'Top model'`

Applies only if the system target file (`SystemTargetFile`) is set to an ERT based system target file (for example, `ert.tlc`). Requires Embedded Coder license.

Specify one of the following values:

- `'Model reference'`: Code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.

- `'Top model'`: Code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

Example: `'CodeInterface','Top model'`

`ObfuscateCode` — Option to obfuscate generated code
true (default) | false

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

Example: `'ObfuscateCode',true`

`Path` — Folder for protected model
current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.

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

`Report` — Option to generate a report
false (default) | true

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

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: `'Report',true`

`OutputFormat` — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

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

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

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

Example: `'OutputFormat','AllReferencedHeaders'`

`Webview` — Option to include a Web view

false (default) | true

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

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdlref_counter`, you can call:

  ```
  Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');
  ```

- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

`Encrypt` — Option to encrypt protected model

false (default) | true

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

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

Example: `'Encrypt',true`

`CustomPostProcessingHook` — Option to add postprocessing function for protected model files
function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. It also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

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

`Modifiable` — Option to create a modifiable protected model
false (default) | true

Option to create a modifiable protected model, specified as a Boolean value. To use this option:

- Add a password for modification using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. If a password has not been added at the time that you create the modifiable protected model, you are prompted to create one.
- Modify the options of your protected model by first providing the modification password using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. Then use the `Simulink.ModelReference.modifyProtectedModel` function to make your option changes.

Example: `'Modifiable',true`

`Callbacks` — Option to specify protected model callbacks
cell array

Option to specify callbacks for a protected model, specified as a cell array of `Simulink.ProtectedModel.Callback` objects.

Example: `'Callbacks',{pmcallback_sim, pmcallback_cg}`

Output Arguments

`harnessHandle` — Handle of the harness model
double

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

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

`neededVars` — Names of base workspace variables
cell array

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

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

Alternatives

"Create a Protected Model"

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords` |
`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel` |
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForModify` |
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` |
`Simulink.ModelReference.ProtectedModel.setPasswordForView` |
`Simulink.ModelReference.modifyProtectedModel`

Topics

Protected Models for Model Reference
"Test the Protected Model"
"Package a Protected Model"
"Specify Custom Obfuscator for Protected Model"

“Configure and Run SIL Simulation” (Embedded Coder)
“Define Callbacks for Protected Models”
“Reference Protected Models from Third Parties” (Simulink)
“Protect Models for Third-Party Use”
“Protected Model File”
“Harness Model”
“Protected Model Report”
“Code Generation Support in Protected Models”
“Code Interfaces for SIL and PIL” (Embedded Coder)

Introduced in R2012b

Simulink.ModelReference.ProtectedModel.clearPasswords

Clear all cached passwords for protected models

Syntax

```
Simulink.ModelReference.ProtectedModel.clearPasswords()
```

Description

`Simulink.ModelReference.ProtectedModel.clearPasswords()` clears all protected model passwords that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear all cached passwords for protected models

After using protected models, clear passwords cached for the models during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswords()
```

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel`

Topics

“Protect Models for Third-Party Use”

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.-clearPasswordsForModel

Clear cached passwords for a protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)
```

Description

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)` clears all protected model passwords for `model` that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear all cached passwords for a protected model

After using a protected model, clear passwords cached for the model during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)
```

Input Arguments

`model` — Protected model name
string or character vector

Model name specified as a string or character vector

Example: '`rtwdemo_counter`'

Data Types: `char`

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords`

Topics

“Protect Models for Third-Party Use”

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.HookInfo class

Package: Simulink.ModelReference.ProtectedModel

Represent files and exported symbols generated by creation of protected model

Description

Specifies information about files and symbols generated when creating a protected model. The creator of a protected model can use this information for postprocessing of the generated files prior to packaging. Information includes:

- List of source code files (`*.c`, `*.h`, `*.cpp`,`*.hpp`).
- List of other related files (`*.mat`, `*.rsp`, `*.prj`, etc.).
- List of exported symbols that you must not modify.

Construction

To access the properties of this class, use the '`CustomPostProcessingHook`' option of the `Simulink.ModelReference.protect` function. The value for the option is a handle to a postprocessing function accepting a `Simulink.ModelReference.ProtectedModel.HookInfo` object as input.

Properties

`ExportedSymbols` — Exported Symbols

cell array of character vectors

A list of exported symbols generated by protected model that you must not modify. Default value is empty.

For a protected model with a top model interface, the `HookInfo` object cannot provide information on exported symbols.

`NonSourceFiles` — Non source code files
cell array of character vectors

A list of non-source files generated by protected model creation. Examples are `*.mat`, `*.rsp`, and `*.prj`. Default value is empty.

`SourceFiles` — Source code files
cell array of character vectors

A list of source code files generated by protected model creation. Examples are `*.c`, `*.h`, `*.cpp`, and `*.hpp`. Default value is empty.

Copy Semantics

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

See Also

`Simulink.ModelReference.protect`

Topics

“Specify Custom Obfuscator for Protected Model”

Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration

Add or provide encryption password for code generation from protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(
model,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(model,password)` adds an encryption password for code generation if you create a protected model. If you use a protected model, the function provides the required password to generate code from the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for code generation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode','Code Generation','Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for code generation.

Generate Code from an Encrypted Protected Model

Use a protected model with encryption for code generation.

Provide the encryption password required for code generation from the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you can generate code from the protected model.

Input Arguments

`model` — Model name
string or character vector

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

`password` — Password for protected model code generation
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for code generation, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForModify

Add or provide password for modifying protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(model,
password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForModify(model, password)` adds a password for a modifiable protected model. After the password has been created, the function provides the password for modifying the protected model.

Examples

Add Functionality to Protected Model

Create a modifiable protected model with support for code generation, then modify it to add Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter', 'password');
```

Add support for Web view to the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Webview',true,...
'Report',true);
```

Input Arguments

`model` — Model name

string or character vector

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

`password` — Password to modify protected model

string or character vector

Password, specified as a string or character vector. The password is required for modification of the protected model.

See Also

`Simulink.ModelReference.modifyProtectedModel` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

Add or provide encryption password for simulation of protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(
model,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model,password)` adds an encryption password for simulation if you create a protected model. If you use a protected model, the function provides the required password to simulate the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for simulation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for simulation.

Simulate an Encrypted Protected Model

Use a protected model with encryption for simulation.

Provide the encryption password required for simulation of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you can simulate the protected model.

Input Arguments

`model` — Model name

string or character vector

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

`password` — Password for protected model simulation

string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for simulation, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForView` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForView

Add or provide encryption password for read-only view of protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(model,
password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForView(model, password)` adds an encryption password for read-only view if you create a protected model. If you use a protected model, the function provides the required password for a read-only view of the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for read-only view.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Webview',true,'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for read-only view.

View an Encrypted Protected Model

Use a protected model with encryption for read-only view.

Provide the encryption password required for the read-only view of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you have access to the read-only view of the protected model.

Input Arguments

`model` — Model name
string or character vector

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

`password` — Password for read-only view of protected model
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for read-only view, the password is required.

See Also

`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.protect`

Introduced in R2014b

Simulink.ProtectedModel.addTarget

Add code generation support for current target to protected model

Syntax

```
Simulink.ProtectedModel.addTarget(model)
```

Description

`Simulink.ProtectedModel.addTarget(model)` adds code generation support for the current `model` target to a protected model of the same name. Each target that the protected model supports is identified by the root of the **Code Generation > System Target file** (`SystemTargetFile`) parameter. For example, if the **System Target file** is `ert.tlc`, the target identifier is `ert`.

To add the current target:

- The model and the protected model of the same name must be on the MATLAB path.
- The protected model must have the `Modifiable` option enabled and have a password for modification.
- The target must be unique in the protected model.

If you add a target to a protected model that did not previously support code generation, the software switches the protected model `Mode` to `CodeGeneration` and `ObfuscateCode` to `true`.

Examples

Add a Target to a Protected Model

Add the currently configured model target to the protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get a list of targets that the protected model supports.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the unprotected model to support a new target.

```
 set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
 save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
 Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`model` — Model name
string or character vector

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

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` |

`Simulink.ProtectedModel.getSupportedTargets` |
`Simulink.ProtectedModel.removeTarget` |
`Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create a Protected Model with Multiple Targets”

Introduced in R2015a

Simulink.ProtectedModel.Callback class

Package: Simulink.ProtectedModel

Represents callback code that executes in response to protected model events

Description

For a protected model functionality, the `Simulink.ProtectedModel.Callback` object specifies code to execute in response to an event. The callback code can be a character vector of MATLAB commands or a MATLAB script. The object includes:

- The code to execute for the callback.
- The event that triggers the callback.
- The protected model functionality that the event applies to.
- The option to override the protected model build.

When you create a protected model, to specify callbacks, call the Simulink.ModelReference.protect on page 2-210 function with the `'Callbacks'` option. The value of this option is a cell array of `Simulink.ProtectedModel.Callback` objects.

Construction

`pmCallback = Simulink.ProtectedModel.Callback(event,functionality, callbackText)` creates a callback object for a specific protected model functionality and event. The `callbackText` specifies MATLAB commands to execute for the callback.

`pmCallback = Simulink.ProtectedModel.Callback(event,functionality, callbackFile)` creates a callback object for a specific protected model functionality and event. The `callbackFile` specifies a MATLAB script to execute for the callback. The script must be on the MATLAB path.

Input Arguments

`event` — Event that triggers callback

`'PreAccess'` | `'Build'`

Callback trigger event. Specify one of the following values:

- `'PreAccess'`: Callback code is executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code is executed before build. Valid only for `'CODEGEN'` functionality.

`functionality` — Protected model functionality

`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Specify one of the following values:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If you do not specify a functionality, the default behavior is `'AUTO'`.

`callbackText` — Callback code to execute

string or character vector

MATLAB commands to execute in response to an event, specified as a string or character vector.

`callbackFile` — Callback script to execute

string or character vector

MATLAB script to execute in response to an event, specified as a string or character vector. Script must be on the MATLAB path.

Properties

`AppliesTo` — Protected model functionality
`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Value is one of the following:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If you do not specify a functionality, the default behavior is `'AUTO'`.

`CallbackFileName` — Callback script to execute
string or character vector

MATLAB script to execute in response to an event, specified as a string or character vector. Script must be on the MATLAB path.

Example: `'pmCallback.m'`

`CallbackText` — Callback code to execute
string or character vector

MATLAB commands to execute in response to an event, specified as a string or character vector.

Example: `'A = [15 150];disp(A)'`

`Event` — Event that triggers callback
`'PreAccess'` | `'Build'`

Callback trigger event. Value is one of the following:

- `'PreAccess'`: Callback code is executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code is executed before build. Valid only for `'CODEGEN'` functionality.

`OverrideBuild` — Option to override protected model build
`false` (default) | `true`

Option to override the protected model build process, specified as a Boolean value. Applies only to a callback object that you define for a `'Build'` event for `'CODEGEN'` functionality. You set this option using the `setOverrideBuild` method.

Methods

setOverrideBuild	Specify option to override protected model build

Copy Semantics

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

Examples

Create Protected Model Using a Callback

Create a callback object with a character vector of MATLAB commands for the callback code. Specify the object when you create a protected model.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess',...
'SIM','disp(''Hello world!'')')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Callbacks',{pmCallback})
sim('sldemo_mdlref_basic')
```

For each instance of the protected model reference in the top model, the output is listed.

```
Hello world!
Hello world!
Hello world!
```

Create Protected Model With a Callback Script

Create a callback object with a MATLAB script for the callback code. Specify the object when you create a protected model.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
rtwbuild('sldemo_mdlref_basic')
```

Before the protected model build process begins, code in `pm_callback.m` executes.

See Also

`Simulink.ModelReference.protect` |
`Simulink.ProtectedModel.getCallbackInfo`

Topics

“Define Callbacks for Protected Models”
“Protect Models for Third-Party Use”
“Code Generation Support in Protected Models”

Introduced in R2016a

setOverrideBuild

Class: Simulink.ProtectedModel.Callback
Package: Simulink.ProtectedModel

Specify option to override protected model build

Syntax

```
setOverrideBuild(override)
```

Description

`setOverrideBuild(override)` specifies whether a `Simulink.ProtectedModel.Callback` object can override the build process. This method is valid only for callbacks that execute in response to a `'Build'` event for `'CODEGEN'` functionality.

Input Arguments

`override` — Option to override protected model build process

false (default) | true

Option to override the protected model build process, specified as a Boolean value. This option applies only to a callback object defined for a `'Build'` event for `'CODEGEN'` functionality.

Example: `pmcallback.setOverrideBuild(true)`

Examples

Create Code Generation Callback to Override Build Process

Create a callback object with a character vector of MATLAB commands for the callback code. Specify that the callback override the build process.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN','disp(''Hello world!'')')
pmCallback.setOverrideBuild(true);
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
rtwbuild('sldemo_mdlref_basic')
```

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.Callback`

Topics

“Define Callbacks for Protected Models”
“Protect Models for Third-Party Use”
“Code Generation Support in Protected Models”

Introduced in R2016a

Simulink.ProtectedModel.CallbackInfo class

Package: Simulink.ProtectedModel

Protected model information for use in callbacks

Description

A `Simulink.ProtectedModel.CallbackInfo` object contains information about a protected model that you can use in the code executed for a callback. The object provides:

- Model name.
- List of models and submodels in the protected model container.
- Callback event.
- Callback functionality.
- Code interface.
- Current target. This information is available only for code generation callbacks.

Construction

`cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event,functionality)` creates a `Simulink.ProtectedModel.CallbackInfo` object.

Properties

`CodeInterface` — Code interface generated by protected model

`'Top model'` | `'Model reference'`

Code interface that the protected model generates.

`Event` — Event that triggered callback

`'PreAccess'` | `'Build'`

Callback trigger event. Value is one of the following:

- `'PreAccess'`: Callback code executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code executed before build. Valid only for `'CODEGEN'` functionality.

`Functionality` — Protected model functionality
`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Value is one of the following:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If the value of `functionality` is blank, the default behavior is `'AUTO'`.

`ModelName` — Protected model name
character vector

Protected model name, specified as a character vector.

`SubModels` — Models and submodels in the protected model container
cell array of character vectors

Names of all models and submodels in the protected model container, specified as a cell array of character vectors.

`Target` — Current target
character vector

Current target identifier for the protected model, specified as a character vector. This property is available only for code generation callbacks.

Methods

getBuildInfoForModel	Get build information object for specified model

Copy Semantics

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

Examples

Use Protected Model Information in Simulation Callback

Create a protected model callback that uses information from the `Simulink.ProtectedModel.Callback` object.

First, on the MATLAB path, create a callback script , `pm_callback.m`, containing:

```
s1 = 'Simulating protected model: ';
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','PreAccess','SIM');
disp([s1 cbinfobj.ModelName])
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('PreAccess'...
,'SIM', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Callbacks',{pmCallback})
```

Simulate the protected model. For each instance of the protected model reference in the top model, the output from the callback is listed.

```
sim('sldemo_mdlref_basic')
```

```
Simulating protected model: sldemo_mdlref_counter
Simulating protected model: sldemo_mdlref_counter
Simulating protected model: sldemo_mdlref_counter
```

See Also

`Simulink.ModelReference.protect` |
`Simulink.ProtectedModel.getCallbackInfo`

Topics

“Define Callbacks for Protected Models”
“Protect Models for Third-Party Use”
“Code Generation Support in Protected Models”

Introduced in R2016a

Simulink.ProtectedModel.getCallbackInfo

Get `Simulink.ProtectedModel.CallbackInfo` object for use by callbacks

Syntax

```
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event,
functionality)
```

Description

`cbinfobj = Simulink.ProtectedModel.getCallbackInfo(modelName,event, functionality)` returns a `Simulink.ProtectedModel.CallbackInfo` object that provides information for protected model callbacks. The object contains information about the protected model, including:

- Model name.
- List of models and submodels in the protected model container.
- Callback event.
- Callback functionality.
- Code interface.
- Current target. This information is available only for code generation callbacks.

Examples

Use Protected Model Information in Code Generation Callback

On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
s1 = 'Code interface is: ';
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
disp([s1 cbinfobj.CodeInterface]);
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

Build the protected model. Before the start of the protected model build process, the code interface is displayed.

```
rtwbuild('sldemo_mdlref_basic')
```

Input Arguments

`modelName` — Protected model name

string or character vector

Protected model name, specified as a string or character vector.

`event` — Event that triggered callback

`'PreAccess'` | `'Build'`

Callback trigger event. Value is one of the following:

- `'PreAccess'`: Callback code executed before simulation, build, or read-only viewing.
- `'Build'`: Callback code executed before build. Valid only for `'CODEGEN'` functionality.

`functionality` — Protected model functionality

`'CODEGEN'` | `'SIM'` | `'VIEW'` | `'AUTO'`

Protected model functionality that the event applies to. Value is one of the following:

- `'CODEGEN'`: Code generation.
- `'SIM'`: Simulation.
- `'VIEW'`: Read-only Web view.
- `'AUTO'`: If the event is `'PreAccess'`, the callback executes for each functionality. If the event is `'Build'`, the callback executes only for `'CODEGEN'` functionality.

If the value of `functionality` is blank, the default behavior is `'AUTO'`.

Output Arguments

`cbinfobj` — Callback information object
`Simulink.ProtectedModel.CallbackInfo`

Callback information, specified as a `Simulink.ProtectedModel.CallbackInfo` object.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

Topics

"Define Callbacks for Protected Models"
"Protect Models for Third-Party Use"
"Code Generation Support in Protected Models"

Introduced in R2016a

getBuildInfoForModel

Class: Simulink.ProtectedModel.CallbackInfo
Package: Simulink.ProtectedModel

Get build information object for specified model

Syntax

```
bldobj = getBuildInfoForModel(model)
```

Description

`bldobj = getBuildInfoForModel(model)` returns a handle to an `RTW.BuildInfo` object. This object specifies the build toolchain and arguments. The `model` name must be in the list of model names in the `SubModels` property of the `Simulink.ProtectedModel.CallbackInfo` object. You can call this method only for code generation callbacks in response to a `'Build'` event.

Input Arguments

`model` — Model name

string or character vector

Model name, specified as a string or character vector. The `model` name must be in the list of model names in the `SubModels` property of the `Simulink.ProtectedModel.CallbackInfo` object. You can call this method only for code generation callbacks in response to a `'Build'` event.

Output Arguments

`bldobj` — Object for build toolchain and arguments

`RTW.BuildInfo`

Build toolchain and arguments, specified as a `RTW.BuildInfo` object. If you do not call the method for a code generation callback and `'Build'` event, the return value is an empty array.

Examples

Get Build Information from a Code Generation Callback

On the MATLAB path, create a callback script, `pm_callback.m`, containing:

```
cbinfobj = Simulink.ProtectedModel.getCallbackInfo(...
'sldemo_mdlref_counter','Build','CODEGEN');
bldinfo = cbinfobj.getBuildInfoForModel(cbinfobj.ModelName);
buildargs = getBuildArgs(bldinfo)
```

When you create a protected model with a simulation callback, use the script.

```
pmCallback = Simulink.ProtectedModel.Callback('Build',...
'CODEGEN', 'pm_callback.m')
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode', 'CodeGeneration','Callbacks',{pmCallback})
```

Build the protected model. Before the start of the protected model build, the build arguments are displayed.

```
rtwbuild('sldemo_mdlref_basic')
```

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.CallbackInfo`

Topics

“Define Callbacks for Protected Models”
“Protect Models for Third-Party Use”
“Code Generation Support in Protected Models”

Introduced in R2016a

Simulink.ProtectedModel.getConfigSet

Get configuration set for current protected model target or for specified target

Syntax

```
configSet = Simulink.ProtectedModel.getConfigSet(protectedModel)
configSet = Simulink.ProtectedModel.getConfigSet(protectedModel,
targetID)
```

Description

`configSet = Simulink.ProtectedModel.getConfigSet(protectedModel)` returns the configuration set object for the current, protected model target.

`configSet = Simulink.ProtectedModel.getConfigSet(protectedModel, targetID)` returns the configuration set object for a specified target that the protected model supports.

Examples

Get Configuration Set for Current Target

Get the configuration set for the currently configured, protected model target.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get the configuration set for the currently configured target.

```
cs = Simulink.ProtectedModel.getConfigSet('mdlref_counter')
```

Get Configuration Set for Specified Target

Get the configuration set for a specified target that the protected model supports.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
 set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
 save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
 Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
 st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Get the configuration set for the added target.

```
cs = Simulink.ProtectedModel.getConfigSet('mdlref_counter','ert')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target identifier
string or character vector

Identifier for selected target, specified as a string or character vector. The target identifier is the root of the **Code Generation > System Target file** (`SystemTargetFile`) parameter. For example, if the **System Target file** is `ert.tlc`, the target identifier is `ert`.

Output Arguments

`configSet` — Configuration object
`Simulink.ConfigSet`

Configuration set, specified as a `Simulink.ConfigSet` object

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget` | `Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create a Protected Model with Multiple Targets”
“Use a Protected Model with Multiple Targets”

Introduced in R2015a

Simulink.ProtectedModel.getCurrentTarget

Get current protected model target

Syntax

```
currentTarget = Simulink.ProtectedModel.getCurrentTarget(
protectedModel)
```

Description

`currentTarget = Simulink.ProtectedModel.getCurrentTarget(protectedModel)` returns the target identifier for the target that is currently configured for the protected model. At the start of a MATLAB session, the default current target is the last target added to the protected model. Otherwise, the current target is the last target that you used. You can change the current target using the `Simulink.ProtectedModel.setCurrentTarget` function.

When building the model, the software changes the target to match the parent if the currently selected target does not match the target of the parent model.

Examples

Get Currently Configured Target for Protected Model

Add a target to a protected model, and then get the currently configured target for the protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Get the currently configured target for the protected model.

```
ct = Simulink.ProtectedModel.getCurrentTarget('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

Output Arguments

`currentTarget` — Current target
character vector

Current target for protected model, specified as a character vector.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget` | `Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create a Protected Model with Multiple Targets”
“Use a Protected Model with Multiple Targets”

Introduced in R2015a

Simulink.ProtectedModel.getSupportedTargets

Get list of targets that protected model supports

Syntax

```
supportedTargets = Simulink.ProtectedModel.getSupportedTargets(
protectedModel)
```

Description

`supportedTargets = Simulink.ProtectedModel.getSupportedTargets(protectedModel)` returns a list of target identifiers for the code generation targets supported by the specified protected model. The target identifier `sim` represents simulation support.

Examples

Get List of Supported Targets for a Protected Model

Add a target to a protected model, and then get a list of supported targets to verify the addition of the new target.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

Output Arguments

`supportedTargets` — List of target identifiers
cell array of character vectors

List of target identifiers for the targets that the protected model supports, specified as a cell array of character vectors.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` |

`Simulink.ProtectedModel.removeTarget` |
`Simulink.ProtectedModel.setCurrentTarget`

Topics

“Create a Protected Model with Multiple Targets”
“Use a Protected Model with Multiple Targets”

Introduced in R2015a

Simulink.ProtectedModel.open

Open protected model

Syntax

```
Simulink.ProtectedModel.open(model)
Simulink.ProtectedModel.open(model,type)
```

Description

`Simulink.ProtectedModel.open(model)` opens a protected model. If you do not specify how to view the protected model, the software first tries to open the Web view. If the Web view is not enabled for the protected model, the software then tries to open the report. If you did not create a report, the software reports an error.

`Simulink.ProtectedModel.open(model,type)` opens a protected model using the specified viewing method. If you specify 'webview', the software opens the Web view for the protected model. If you specify 'report', the software opens the protected model report. If the method that you specify is not enabled, the software reports an error. The protected model is not opened.

Examples

Open a Protected Model

Open a protected model with no specified method.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model enabling support for code generation and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Report',true);
```

Open the protected model without specifying how to view it.

```
Simulink.ProtectedModel.open('mdlref_counter')
```

The protected model does not have Web view enabled, so the protected model report is opened.

Open a Protected Model Web View

Open a protected model, specifying the Web view.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model with support for code generation, Web view, and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Webview',true,'Report',true);
```

Open the protected model and specify that you want to see the Web view.

```
Simulink.ProtectedModel.open('mdlref_counter','webview')
```

The protected model Web view is opened.

Input Arguments

`model` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`type` — Open method
‘webview‘ | ‘report‘

Method for viewing the protected model. If you specify ‘webview’, the software opens the Web view for the protected model. If you specify ‘report’, the software opens the protected model report.

See Also

`Simulink.ModelReference.protect`

Introduced in R2015a

Simulink.ProtectedModel.removeTarget

Remove support for specified target from protected model

Syntax

```
Simulink.ProtectedModel.removeTarget(protectedModel,targetID)
```

Description

`Simulink.ProtectedModel.removeTarget(protectedModel,targetID)` removes code generation support for the specified target from a protected model. You must provide the modification password to make this update. Removing a target does not require access to the unprotected model.

Note You cannot remove the `sim` target. If you do not want the protected model to support simulation, use the `Simulink.ModelReference.modifyProtectedModel` function to change the protected model mode to `ViewOnly`.

Examples

Remove Target Support from a Protected Model

Remove a supported target from a protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Remove support for the `ert` target from the protected model. You are prompted for the modification password.

```
Simulink.ProtectedModel.removeTarget('mdlref_counter','ert');
```

Verify that support for the `ert` target has been removed from the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target to be removed
string or character vector

Identifier for target to be removed, specified as a string or character vector.

See Also

`Simulink.ModelReference.modifyProtectedModel` | `Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.setCurrentTarget`

Topics

"Create a Protected Model with Multiple Targets"

Introduced in R2015a

Simulink.ProtectedModel.setCurrentTarget

Configure protected model to use specified target

Syntax

```
Simulink.ProtectedModel.setCurrentTarget(protectedModel, targetID)
```

Description

`Simulink.ProtectedModel.setCurrentTarget(protectedModel, targetID)` configures the protected model to use the target that the target identifier specifies.

Note If you include the protected model in a model reference hierarchy, the software tries to change the current target to match the target of the parent model. If the software cannot match the target of the parent, it reports an error.

Examples

Set Current Target for Protected Model

After you get a list of supported targets, set the current target for a protected model.

Load the model and save a local copy.

```
sldemo_mdlref_counter
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Add a required password for modifying a protected model. If you do not add a password, you are prompted to set a password when you create a modifiable, protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'mdlref_counter','password');
```

Create a modifiable, protected model with support for code generation.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Get a list of targets that the protected model supports.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the unprotected model to support a new target.

```
set_param('mdlref_counter', 'SystemTargetFile', 'ert.tlc');
save_system('mdlref_counter');
```

Add support to the protected model for the new target. You are prompted for the modification password.

```
Simulink.ProtectedModel.addTarget('mdlref_counter');
```

Verify that support for the new target has been added to the protected model.

```
st = Simulink.ProtectedModel.getSupportedTargets('mdlref_counter')
```

Configure the protected model to use the new target.

```
Simulink.ProtectedModel.setCurrentTarget('mdlref_counter','ert');
```

Verify that the current target is correct.

```
ct = Simulink.ProtectedModel.getCurrentTarget('mdlref_counter')
```

Input Arguments

`protectedModel` — Model name

string or character vector

Protected model name, specified as a string or character vector.

`targetID` — Target identifier

string or character vector

Identifier for selected target, specified as a string or character vector.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget`

Topics

“Create a Protected Model with Multiple Targets”
“Use a Protected Model with Multiple Targets”

Introduced in R2015a

slConfigUIGetVal

Return current value for custom target configuration option

Syntax

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

Input Arguments

`hDlg`

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

`hSrc`

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

`'OptionName'`

Quoted name of the TLC variable defined for a custom target configuration option.

Output Arguments

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

Description

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

Examples

In the following example, the `slConfigUIGetVal` function returns the value of the **Configuration Parameters > Code Generation > Interface > Advanced parameters > Terminate function required** option.

```
function usertarget_selectcallback(hDlg,hSrc)

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

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

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

See Also

`slConfigUISetEnabled` | `slConfigUISetVal`

Topics

“Define and Display Custom Target Options”
“Custom Target Optional Features”

Introduced in R2006b

slConfigUISetEnabled

Enable or disable custom target configuration option

Syntax

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

Arguments

`hDlg`

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

`hSrc`

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

`'OptionName'`

Quoted name of the TLC variable defined for a custom target configuration option.

`true`

Specifies that the option should be enabled.

`false`

Specifies that the option should be disabled.

Description

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

If you use this function to disable a parameter that is represented in the Configuration Parameters dialog box, the parameter appears greyed out in the dialog context.

Examples

In the following example, the `slConfigUISetEnabled` function disables the **Configuration Parameters > Code Generation > Interface > Advanced parameters > Terminate function required** option.

```
function usertarget_selectcallback(hDlg,hSrc)

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

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

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

See Also

`slConfigUIGetVal` | `slConfigUISetVal`

Topics

“Define and Display Custom Target Options”
“Custom Target Optional Features”

Introduced in R2006b

slConfigUISetVal

Set value for custom target configuration option

Syntax

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

Arguments

`hDlg`

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

`hSrc`

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

`'OptionName'`

Quoted name of the TLC variable defined for a custom target configuration option.

`OptionValue`

Value to be set for the specified option.

Description

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

Examples

In the following example, the `slConfigUISetVal` function sets the value `'off'` for the **Configuration Parameters > Code Generation > Interface > Advanced parameters > Terminate function required** option.

```
function usertarget_selectcallback(hDlg,hSrc)

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

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

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

See Also

`slConfigUIGetVal` | `slConfigUISetEnabled`

Topics

“Define and Display Custom Target Options”
“Custom Target Optional Features”

Introduced in R2006b

switchTarget

Select target for model configuration set

Syntax

```
switchTarget(myConfigObj,systemTargetFile,[])
switchTarget(myConfigObj,systemTargetFile,targetOptions)
```

Description

`switchTarget(myConfigObj,systemTargetFile,[])` changes the selected system target file for the active configuration set.

`switchTarget(myConfigObj,systemTargetFile,targetOptions)` sets the configuration parameters specified by `targetOptions`.

Examples

Get ConfigSet, Default Options, and Switch Target

This example shows how to get the active configuration set for `model`, and change the system target file for the configuration set.

```
% Get configuration set for model
myConfigObj = getActiveConfigSet(model);
% Switch system target file
switchTarget(myConfigObj,'ert.tlc',[]);
```

Get ConfigSet, Set Options, Switch Target

This example shows how to get the active configuration set for the current model (`gcs`), set various `targetOptions`, then change the system target file selection.

```
% Get configuration set for current model
myConfigObj=getActiveConfigSet(gcs);

% Specify target options
targetOptions.TLCOptions = '-aVarName=1';
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = 'my target';
targetOptions.TemplateMakefile = 'grt_default_tmf';

% Define a system target file
targetSystemFile='grt.tlc';

% Switch system target file
switchTarget(myConfigObj,targetSystemFile,targetOptions);
```

Use `targetOptions` to verify values (optional).

```
% Verify values (optional)
targetOptions
```

```
          TLCOptions: '-aVarName=1'
         MakeCommand: 'make_rtw'
         Description: 'my target'
    TemplateMakefile: 'grt_default_tmf'
```

Get ConfigSet, Set Options for MSVC Solution Build, Switch Target to MSVC ERT

This example shows how to get the active configuration set for `model`, then change the system target file to the ERT Create Visual C/C++ Solution File for Embedded Coder.

```
model='rtwdemo_rtwintro';
open_system(model);

% Get configuration set for model
myConfigObj = getActiveConfigSet(model);

% Specify target options for MSVC build
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = ...
     'Create Visual C/C++ Solution File for Embedded Coder';
targetOptions.TemplateMakefile = 'RTW.MSVCBuild';
```

```
% Switch system target file
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Get ConfigSet, Set Options for Toolchain Build, and Switch Target

Use options to select default ERT target file, instead of `set_param(model,'SystemTargetFile','ert.tlc')`.

```
% use switchTarget to select toolchain build of defaul ERT target
model='rtwdemo_rtwintro';
open_system(model);

% Get configuration set for model
myConfigObj = getActiveConfigSet(model);

% Specify target options for toolchain build approach
targetOptions.MakeCommand = '';
targetOptions.Description = 'Embedded Coder';
targetOptions.TemplateMakefile = '';

% Switch system target file
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Input Arguments

`myConfigObj` — Configuration set object

`object`

A configuration set object of `Simulink.ConfigSet` or configuration reference object of `Simulink.ConfigSetRef`. Call `getActiveConfigSet` to get the configuration set object.

Example: `myConfigObj = getActiveConfigSet(model);`

`systemTargetFile` — Name of system target file

character vector

Specify the name of the system target file (such as `ert.tlc` for Embedded Coder or `grt.tlc` for Simulink Coder) as the name appears in the **System Target File Browser**.

Example: `systemTargetFile = 'ert.tlc';`

`targetOptions` — Structure with field values that provide configuration parameter options
struct

Structure with fields that define a code generation target options. You can choose to modify certain configuration parameters by filling in values in a structure field. If you do not want to use options, specify an empty structure (`[]`).

Field Values in targetOptions

Specify the structure field values of the `targetOptions`. For no options, specify an empty structure (`[]`).

Example: `targetOptions = [];`

`TemplateMakefile` — Character vector specifying file name of template makefile
character vector

Example: `targetOptions.TemplateMakefile = 'RTW.MSVCBuild';`

`TLCOptions` — Character vector specifying TLC argument
character vector

Example: `targetOptions.TLCOptions = '-aVarName=1';`

`MakeCommand` — Character vector specifying make command MATLAB language file
character vector

Example: `targetOptions.MakeCommand = 'make_rtw';`

`Description` — Character vector specifying description of the system target file
character vector

Example: `targetOptions.Description = 'Create Visual C/C++ Solution File for Embedded Coder';`

See Also

`Simulink.ConfigSet` | `Simulink.ConfigSetRef` | `getActiveConfigSet`

Topics

“Select a System Target File Programmatically”

“Configure a System Target File”
“Set Target Language Compiler Options”

Introduced in R2009b

tlc

Invoke Target Language Compiler to convert model description file to generated code

Syntax

`tlc [-options] [file]`

Description

`tlc [-options] [file]` invokes the Target Language Compiler (TLC) from the command prompt. The TLC converts the model description file, *`model`*`.rtw` (or similar files), into target-specific code or text. Typically, you do not call this command because the build process automatically invokes the Target Language Compiler when generating code. For more information, see “Target Language Compiler Basics”.

Note This command is used only when invoking the TLC separately from the build process. You cannot use this command to initiate code generation for a model.

You can change the default behavior by specifying one or more compilation *options* as described in “Options” on page 2-282

Options

You can specify one or more compilation options with each `tlc` command. Use spaces to separate options and arguments. TLC resolves options from left to right. If you use conflicting options, the right-most option prevails. The `tlc` options are:

-r Specify *`model`*`.rtw` file name

`-r` *file_name*

Specify the file name that you want to translate.

-v Specify verbose level

`-v` *number*

Specify a `number` indicating the verbose level. If you omit this option, the default value is one.

-l Specify path to local include files

`-l` *path*

Specify a folder `path` to local include files. The TLC searches this path in the order specified.

-m Specify maximum number of errors

`-m` *number*

Specify the maximum `number` of errors reported by the TLC prior to terminating the translation of the `.tlc` file.

If you omit this option, the default value is five.

-O Specify the output file path

`-O` *path*

Specify the folder `path` to place output files.

If you omit this option, TLC places output files in the current folder.

-d[a|c|n|o] Invoke debug mode

`-da` execute any %assert directives

`-dc` invoke the TLC command line debugger

`-dn` produce log files, which indicate those lines hit and those lines missed during compilation.

`-do` disable debugging behavior

-a Specify parameters

`-a` *identifier* = *expression*

Specify parameters to change the behavior of your TLC program. For example, this option is used by the code generator to set inlining of parameters or file size limits.

-p Print progress

`-p` *number*

Print a '.' indicating progress for every `number` of TLC primitive operations executed.

-lint Performance checks and runtime statistics

`-lint`

Perform simple performance checks and collect runtime statistics.

-xO Parse only

`-x0`

Parse only a TLC file; do not execute it.

Introduced in R2009a

updateFilePathsAndExtensions

Update files in model build information with missing paths and file extensions

Syntax

```
updateFilePathsAndExtensions(buildinfo,extensions)
```

Description

`updateFilePathsAndExtensions(buildinfo,extensions)` specifies the file name extensions (file types) to include in search and update processing.

Using paths from the build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information require an updated path or file extension. Use this function to:

- Maintain build information for a toolchain that requires the use of file extensions.
- Update multiple customized instances of build information for a given model.

If you use `updateFilePathsAndExtensions`, you call it after you add files to the build information. This approach minimizes the potential performance impact of the required disk I/O.

Examples

Update File Paths and Extensions in Build Information

In your working folder, create the folder path `etcproj/etc` , add files `etc.c`, `test1.c`, and `test2.c` to the folder `etc`. For this example, the working folder is `w:\work \BuildInfo`. From the working folder, update build information `myModelBuildInfo` with missing paths or file extensions.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo,fullfile(pwd, ...
```

```
   'etcproj','/etc'),'test');
addSourceFiles(myModelBuildInfo,{'etc' 'test1' ...
   'test2'},'','test');
before = getSourceFiles(myModelBuildInfo,true,true);


>> before

before =

    '\etc'    '\test1'    '\test2'

updateFilePathsAndExtensions(myModelBuildInfo);
after = getSourceFiles(myModelBuildInfo,true,true);


>> after{:}

ans =

    'w:\work\BuildInfo\etcproj\etc\etc.c'


ans =

    'w:\work\BuildInfo\etcproj\etc\test1.c'


ans =

    'w:\work\BuildInfo\etcproj\etc\test2.c'
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo` object

`extensions` — File name extensions to include in search and update processing `'.c'` (default) | cell array of character vectors | string

The *`extensions`* argument specifies the file name extensions (file types) to include in search and update processing. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For

example, if you specify `{'.c' '.cpp'}` and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` is updated to `myfile.c`.

Example: `'.c' '.cpp'`

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFileSeparator`

Topics

“Customize Post-Code-Generation Build Processing”

Introduced in R2006a

updateFileSeparator

Update file separator character for file lists in model build information

Syntax

```
updateFileSeparator(buildinfo,separator)
```

Description

`updateFileSeparator(buildinfo,separator)` changes instances of the current file separator (/ or \) in the model build information to the specified file separator.

The default value for the file separator matches the value returned by the MATLAB command `filesep`. For template makefile (TMF) approach builds, you can override the default by defining a separator with the `MAKEFILE_FILESEP` macro in the template makefile (see “Cross-Compile Code Generated on Microsoft Windows”. If the `GenerateMakefile` parameter is set, the code generator overrides the default separator and updates the model build information after evaluating the `PostCodeGenCommand` configuration parameter.

Examples

Update File Separator in Build Information

Update object `myModelBuildInfo` to apply the Windows® file separator.

```
myModelBuildInfo = RTW.BuildInfo;
updateFileSeparator(myModelBuildInfo,'\');
```

Input Arguments

`buildinfo` — Name of build information object returned by `RTW.BuildInfo`
object

`separator` — File separator character for path specifications in the build information

`'\' | '/'`

The separator argument specifies the file separator `\` (Windows) or `/` (UNIX®) to use in file path specifications in the build information.

Example: '\'

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFilePathsAndExtensions`

Topics

“Customize Post-Code-Generation Build Processing”
“Cross-Compile Code Generated on Microsoft Windows”

Introduced in R2006a

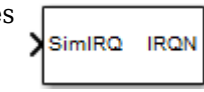
3

Blocks in Simulink Coder—Alphabetical List

Async Interrupt

Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that execute downstream subsystems or Task Sync blocks

Library: Simulink Coder / Asynchronous / Interrupt Templates



Description

For each specified VME interrupt level in the example RTOS (VxWorks®), the Async Interrupt block generates an interrupt service routine (ISR) that calls one of the following:

- A function call subsystem
- A Task Sync block
- A Stateflow chart configured for a function call input event

Note You can use the blocks in the `vxlib1` library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Assumptions and Limitations

- The block supports VME interrupts 1 through 7.
- The block uses these RTOS (VxWorks) system calls:

```
sysIntEnable
sysIntDisable
intConnect
intLock
intUnlock
tickGet
```

Performance Considerations

Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. Usually, it is best to keep ISRs as short as possible. Connect only function-call subsystems that contain a few blocks to an Async Interrupt block.

A better solution for large subsystems is using the Task Sync block to synchronize the execution of the function-call subsystem to an RTOS task. Place the Task Sync block between the Async Interrupt block and the function-call subsystem. The Async Interrupt block then uses the Task Sync block as the ISR. The ISR releases a synchronization semaphore (performs a `semGive`) to the task, and returns immediately from interrupt level. The example RTOS (VxWorks) then schedules and runs the task. See the description of the Task Sync block.

Ports

Input

`Input` — Simulated interrupt source

scalar

A simulated interrupt source.

Output Arguments

`Output` — Control signal

scalar

Control signal for a:

- Function-call subsystem
- Task Sync block
- Stateflow chart configured for a function call input event

Parameters

`VME interrupt number(s)` — VME interrupt numbers for the interrupts to be installed
[1 2] (default) | integer array

An array of VME interrupt numbers for the interrupts to be installed. The valid range is `1..7`.

The width of the Async Interrupt block output signal corresponds to the number of VME interrupt numbers specified.

Note A model can contain more than one Async Interrupt block. However, if you use more than one Async Interrupt block, do not duplicate the VME interrupt numbers specified in each block.

`VME interrupt vector offset(s)` — Interrupt vector offset numbers corresponding to the VME interrupt numbers
[192 193] (default) | integer array

An array of unique interrupt vector offset numbers corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field. The Stateflow software passes the offsets to the RTOS (VxWorks) call `intConnect(INUM_TO_IVEC(offset),...)`.

`Simulink task priority(s)` — Priority of downstream blocks
[10 11] (default) | integer array

The Simulink priority of downstream blocks. Each output of the Async Interrupt block drives a downstream block (for example, a function-call subsystem). Specify an array of priorities corresponding to the VME interrupt numbers that you specify for **VME interrupt number(s)**.

The **Simulink task priority** values are required to generate a rate transition code (see “Rate Transitions and Asynchronous Blocks”). Simulink task priority values are also required to maintain absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. The assigned priorities typically are higher than the priorities assigned to periodic tasks.

Note The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

`Preemption flag(s); preemptable-1; non-preemptable-0` — Selects preemption

[0 1] (default) | integer array

Set this option to 1 if an output signal of the Async Interrupt block drives a Task Sync block.

Higher priority interrupts can preempt lower priority interrupts in the example RTOS (VxWorks). To lock out interrupts during the execution of an ISR, set the pre-emption flag to 0. This setting causes generation of `intLock()` and `intUnlock()` calls at the beginning and end of the ISR code. Use interrupt locking carefully, as it increases the interrupt response time of the system for interrupts at the `intLockLevelSet()` level and below. Specify an array of flags corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field.

Note The number of elements in the arrays specifying **VME interrupt vector offset(s)** and **Simulink task priority** must match the number of elements in the **VME interrupt number(s)** array.

`Manage own timer` — Select timer manager

on (default) | off

If selected, the ISR generated by the Async Interrupt block manages its own timer by reading absolute time from the hardware timer. Specify the size of the hardware timer with the **Timer size** option.

`Timer resolution (seconds)` — Resolution of ISR timer

1/60 (default)

The resolution of the ISRs timer. ISRs generated by the Async Interrupt block maintain their own absolute time counters. By default, these timers obtain their values from the RTOS (VxWorks) kernel by using the `tickGet` call. The **Timer resolution** field determines the resolution of these counters. The default resolution is `1/60` second. The `tickGet` resolution for your board support package (BSP) can be different. Determine the `tickGet` resolution for your BSP and enter it in the **Timer resolution** field.

If you are targeting an RTOS other than the example RTOS (VxWorks), replace the `tickGet` call with an equivalent call to the target RTOS. Or, generate code to read the timer register on the target hardware. For more information, see “Timers in Asynchronous Tasks” and “Async Interrupt Block Implementation”.

`Timer size` — Number of bits to store the clock tick
`32bits` (default) | `16bits` | `8bits` | `auto`

The number of bits to store the clock tick for a hardware timer. The ISR generated by the Async Interrupt block uses the timer size when you select **Manage own timer**. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the code generator determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. When **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, the code generator uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters”. See also “Timers in Asynchronous Tasks”.

`Enable simulation input` — Select add simulation input port
on (default) | off

If selected, the Simulink software adds an input port to the Async Interrupt block. This port is for simulation only. Connect one or more simulated interrupt sources to the simulation input.

Note Before generating code, consider removing blocks that drive the simulation input to prevent the blocks from contributing to the generated code. Alternatively, you can use the Environment Controller block, as explained in “Dual-Model Approach: Code Generation”. If you use the Environment Controller block, the sample times of driving blocks contribute to the sample times supported in the generated code.

See Also

Task Sync

Topics

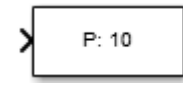
“Asynchronous Events”
“Asynchronous Events”

Introduced in R2006a

Asynchronous Task Specification

Specify priority of asynchronous task represented by referenced model triggered by asynchronous interrupt

Library: Simulink Coder / Asynchronous



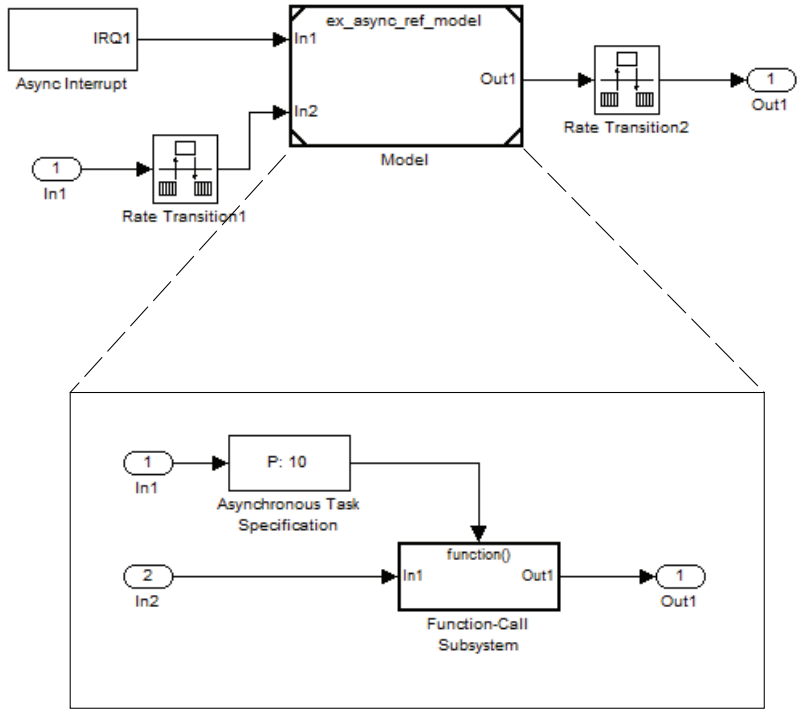
Description

The Asynchronous Task Specification block specifies parameters, such as the task priority, of an asynchronous task represented by a function-call subsystem with a trigger from an asynchronous interrupt. Use this block to control scheduling of function-call subsystems with triggers from asynchronous events. You control the scheduling by assigning a priority to each function-call subsystem within a referenced model.

To use this block, follow the procedure in “Convert an Asynchronous Subsystem into a Model Reference”.

Observe in the figure:

- The block must reside in a referenced model between a root-level Inport block and a function-call subsystem. The Asynchronous Task Specification block must immediately follow and connect directly to the Inport block.
- The Inport block must receive an interrupt signal from an Async Interrupt block that is in the parent model.
- The Inport block must be configured to receive and send function-call trigger signals.



Ports

Input

`Port_1` — Interrupt input signal

scalar

Interrupt input signal received from a root-level Inport block.

Output

`Port_1` — Interrupt signal with priority

scalar

Interrupt signal with specified task priority that triggers a function-call subsystem.

Parameters

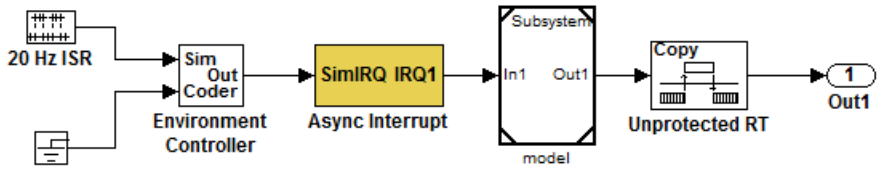
`Task priority` — Priority of asynchronous task that calls function-call subsystem

`10` (default)

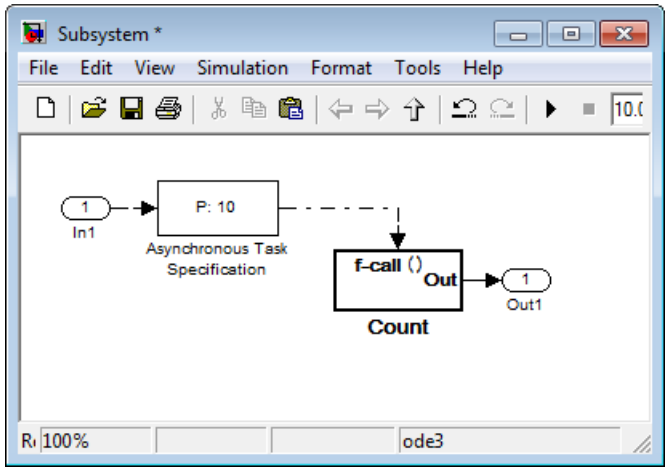
Specify an integer or [] as the priority of the asynchronous task that calls the connected function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

- If you specify an integer, it must match the priority value of the interrupt signal initiator in the parent model.
- If you specify [], the priority does not have to match the priority of the interrupt signal initiator in the top model. The rate transition algorithm is conservative (not optimized). The priority is unknown but static.

Consider the following model.



The referenced model has the following content.



If the **Task priority** parameter is set to 10, the Async Interrupt block in the parent model must also have a priority of 10. If the parameter is set to [], the priority of the Async Interrupt block can be a value other than 10.

See Also

Blocks
Function-Call Subsystem | Inport

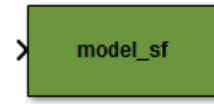
Topics

“Asynchronous Events”
“Spawn and Synchronize Execution of RTOS Task”
“Pass Asynchronous Events in RTOS as Input To a Referenced Model”
“Convert an Asynchronous Subsystem into a Model Reference”
“Rate Transitions and Asynchronous Blocks”
“Asynchronous Support”
“Asynchronous Events”
“Model References” (Simulink)

Introduced in R2011a

Generated S-Function

Represent model or subsystem as generated S-function code
Library: Simulink Coder / S-Function Target



Description

An instance of the Generated S-Function block represents code that the code generator produces from its S-function system target file for a model or subsystem. For example, you extract a subsystem from a model and build a Generated S-Function block from it by using the S-function target. This mechanism can be useful for:

- Converting models and subsystems to application components
- Reusing models and subsystems
- Optimizing simulation—often, an S-function simulates more efficiently than the original model

For details on how to create a Generated S-Function block from a subsystem, see “Create S-Function Blocks from a Subsystem”.

Requirements

- The S-Function block must perform identically to the model or subsystem from which it was generated.
- Before creating the block, explicitly specify Inport block signal attributes, such as signal widths or sample times. The sole exception to this rule concerns sample times, as described in “Sample Time Propagation in Generated S-Functions”.
- Set the solver parameters of the Generated S-Function block to be the same as the parameters of the original model or subsystem. The generated S-function code operates identically to the original subsystem (for an exception to this rule, see “Choose a Solver Type”).

Ports

Input

`Input` — S-function input
varies

See requirements.

Output Arguments

`Output` — S-function output
varies

See requirements.

Parameters

`Generated S-function name (model_sf)` — Name of S-function
model_sf (default) | character vector

The name of the generated S-function. The code generator derives the name by appending `_sf` to the name of the model or subsystem from which the block is generated.

`Show module list` — Select display module list
off (default) | on

If selected, displays modules generated for the S-function.

See Also

Topics

“Generate S-Function from Subsystem”
“Create S-Function Blocks from a Subsystem”

Introduced in R2011b

Model Header

Specify external header code

Description

For a model that includes the Model Header block, the code generator adds external code that you specify to the header file (*`model`*`.h`) that it generates. You can specify code for the code generator to add near the top and bottom of the header file.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`Top of Model Header` — Code to add near top of generated header file

Specify code that you want the code generator to add near the top of the header file for the model. The code generator places the code in the section labeled `user code (top of header file)`.

`Bottom of Model Header` — Code to add at bottom of generated header file

Specify code that you want the code generator to add at the bottom of the header file for the model. The code generator places the code in the section labeled `user code (bottom of header file)`.

See Also

Model Source | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

Model Source

Specify external source code

Description

For a model that includes the Model Source block, the code generator adds external code that you specify to the source file (*`model`*`.c` or *`model`*`.cpp`) that it generates. You can specify code for the code generator to add near the top and bottom of the source file.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`Top of Model Header` — Code to add near top of generated source file

Specify code that you want the code generator to add near the top of the source file for the model. The code generator places the code in the section labeled `user code (top of source file)`.

`Bottom of Model Header` — Code to add at bottom of generated source file

Specify code that you want the code generator to add at the bottom of the source file for the model. The code generator places the code in the section labeled `user code (bottom of source file)`.

Example

See “Add External Code to Generated Start Function”.

See Also

Model Header | System Disable | System Outputs | System Update | System Derivatives | System Enable | System Initialize | System Start | System Terminate

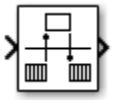
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

Protected RT

Handle transfer of data between blocks operating at different rates and maintain data integrity



Library

VxWorks (`vxlib1`)

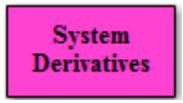
Description

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.

Introduced in R2006a

System Derivatives

Specify external system derivative code



Description

For a model or nonvirtual subsystem that includes the System Derivatives block and a block that computes continuous states, the code generator adds external code, which you specify, to the `SystemDerivatives` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Derivatives Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemDerivatives` function for the model or subsystem.

`System Derivatives Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemDerivatives` function for the model or subsystem.

`System Derivatives Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemDerivatives` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Disable | System Enable | System Outputs | System Start | System Terminate | System Update

Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

System Disable

Specify external system disable code

Description

For a model or nonvirtual subsystem that includes the System Disable block and a block that uses a `SystemDisable` function, the code generator adds external code, which you specify, to the `SystemDisable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Disable Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemDisable` function for the model or subsystem.

`System Disable Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemDisable` function for the model or subsystem.

`System Disable Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemDisable` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Derivatives | System Enable | System Outputs | System Start | System Terminate | System Update

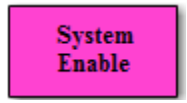
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

System Enable

Specify external system enable code



Description

For a model or nonvirtual subsystem that includes the System Enable block and a block that uses a `SystemEnable` function, the code generator adds external code, which you specify, to the `SystemEnable` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Enable Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemEnable` function for the model or subsystem.

`System Enable Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemEnable` function for the model or subsystem.

`System Enable Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemEnable` function for the model or subsystem.

See Also

Model Header | Model Source | System Initialize | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

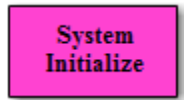
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

System Initialize

Specify external system initialization code



Description

For a model or nonvirtual subsystem that includes the System Initialize block and a block that uses a `SystemInitialize` function, the code generator adds external code, which you specify, to the `SystemInitialize` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Initialize Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemInitialize` function for the model or subsystem.

`System Initialize Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemInitialize` function for the model or subsystem.

`System Initialize Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemInitialize` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Outputs | System Start | System Terminate | System Update

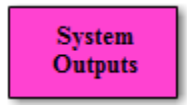
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

System Outputs

Specify external system outputs code



Description

For a model or nonvirtual subsystem that includes the System Outputs block and a block that uses a `SystemOutputs` function, the code generator adds external code, which you specify, to the `SystemOutputs` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Outputs Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemOutputs` function for the model or subsystem.

`System Outputs Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemOutputs` function for the model or subsystem.

`System Outputs Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemOutputs` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Derivatives | System Disable | System Initialize | System Start | System Terminate | System Update

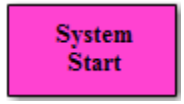
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

System Start

Specify external system startup code



Description

For a model or nonvirtual subsystem that includes the System Start block and a block that uses a `SystemStart` function, the code generator adds external code, which you specify, to the `SystemStart` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Start Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemStart` function for the model or subsystem.

`System Start Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemStart` function for the model or subsystem.

`System Start Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemStart` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Terminate | System Derivatives | System Disable | System Initialize | System Outputs | System Update

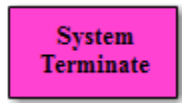
Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

System Terminate

Specify external system termination code



Description

For a model or nonvirtual subsystem that includes the System Terminate block and a block that uses a `SystemTerminate` function, the code generator adds external code, which you specify, to the `SystemTerminate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Terminate Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemTerminate` function for the model or subsystem.

`System Disable Terminate Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemTerminate` function for the model or subsystem.

`System Disable Terminate Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemTerminate` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Update

Topics

“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

System Update

Specify external system update code

Description

For a model or nonvirtual subsystem that includes the System Update block and a block that uses a `SystemUpdate` function, the code generator adds external code, which you specify, to the `SystemUpdate` function that it generates. You can specify code for the code generator to add to the declaration, execution, and exit sections of the function code.

Note If you include this block in a referenced model, the code generator ignores the block for simulation target builds, but processes the block for other system target files.

Parameters

`System Update Function Declaration Code` — Code to add to the declaration section of the generated function

Specify code that you want the code generator to add to the declaration section of the `SystemUpdate` function for the model or subsystem.

`System Update Function Execution Code` — Code to add to the execution section of the generated function

Specify code that you want the code generator to add to the execution section of the `SystemUpdate` function for the model or subsystem.

`System Update Function Exit Code` — Code to add to the exit section of the generated function

Specify code that you want the code generator to add to the exit section of the `SystemUpdate` function for the model or subsystem.

See Also

Model Header | Model Source | System Enable | System Start | System Derivatives | System Disable | System Initialize | System Outputs | System Terminate

Topics

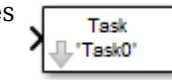
“Place External C/C++ Code in Generated Code” (Embedded Coder)

Introduced in R2006a

Task Sync

Run code of downstream function-call subsystem or Stateflow chart by spawning an example RTOS (VxWorks) task

Library: Simulink Coder / Asynchronous / Interrupt Templates



Description

The Task Sync block spawns an example RTOS (VxWorks) task that calls a function-call subsystem or Stateflow chart. Typically, you place the Task Sync block between an Async Interrupt block and a function-call subsystem block or Stateflow chart. Alternatively, you could connect the Task Sync block to the output port of a Stateflow diagram that has an event, `Output to Simulink`, configured as a function call.

The Task Sync block:

- Uses the RTOS (VxWorks) system call `taskSpawn` to spawn an independent task. When the task is activated, it calls the downstream function-call subsystem code or Stateflow chart. The block calls `taskDelete` to delete the task during model termination.
- Creates a semaphore to synchronize the connected subsystem with execution of the block.
- Wraps the spawned task in an infinite `for` loop. In the loop, the spawned task listens for the semaphore by using `semTake`. The first call to `semTake` specifies `NO_WAIT`. This setting lets the task determine whether a second `semGive` has occurred before the completion of the function-call subsystem or chart. This sequence indicates that the interrupt rate is too fast or the task priority is too low.
- Generates synchronization code (for example, `semGive()`). This code lets the spawned task run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. The connection between the Async Interrupt and Task Sync blocks accomplishes this operation and triggers execution of the Task Sync block within an ISR.
- Supplies absolute time if blocks in the downstream algorithmic code require it. The time comes from the timer maintained by the Async Interrupt block or comes from an independent timer maintained by the task associated with the Task Sync block.

When you design your application, consider when timer and signal input values could be taken for the downstream function-call subsystem that is connected to the Task Sync block. By default, the time and input data are read when the RTOS (VxWorks) activates the task. For this case, the data (input and time) are synchronized to the task itself. If you select the **Synchronize the data transfer of this task with the caller task** option and the Task Sync block driver is an Async Interrupt block, the time and input data are read when the interrupt occurs (that is, within the ISR). For this case, data is synchronized with the caller of the Task Sync block.

Note You can use the blocks in the `vxlib1` library (Async Interrupt and Task Sync) for simulation and code generation. These blocks provide starting point examples to help you develop custom blocks for your target environment.

Ports

Input

`Input` — Call from interrupt block
call

A call from an Async Interrupt block.

Output Arguments

`Output` — Call to function-call subsystem
call

A call to a function-call subsystem.

Parameters

`Task name (10 characters or less)` — Task function name
`Task0` (default) | character vector

The first argument passed to the `taskSpawn` system call in the RTOS. The RTOS (VxWorks) uses this name as the task function name. This name also serves as a

debugging aid. Routines use the task name to identify the task from which they are called.

`Simulink task priority (0–255)` — RTOS task priority
50 (default) | integer

The RTOS task priority assigned to the function-call subsystem task when spawned. RTOS (VxWorks) priorities range from 0 to 255, with 0 representing the highest priority.

Note The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

`Stack size (bytes)` — Maximum size for stack of the task
1024 (default) | integer

Maximum size to which the stack of the task can grow. The stack size is allocated when the RTOS (VxWorks) spawns the task. Choose a stack size based on the number of local variables in the task. Determine the size by examining the generated code for the task (and functions that are called from the generated code).

`Synchronize the data transfer of this task with the caller task` — Select synchronization
off (default) | on

If not selected (the default),

- The block maintains a timer that provides absolute time values required by the computations of downstream blocks. The timer is independent of the timer maintained by the Async Interrupt block that calls the Task Sync block.
- A **Timer resolution** option appears.
- The **Timer size** option specifies the word size of the time counter.

If selected,

- The block does not maintain an independent timer and does not display the **Timer resolution** field.
- Downstream blocks that require timers use the timer maintained by the Async Interrupt block that calls the Task Sync block (see “Timers in Asynchronous Tasks”).

The timer value is read at the time the asynchronous interrupt is serviced. Data transfers to blocks called by the Task Sync block execute within the task associated with the Async Interrupt block. Therefore, data transfers are synchronized with the caller.

`Timer resolution (seconds)` — Resolution for timer of the block
1/60 (default)

The resolution of the timer of the block in seconds. This option appears only if **Synchronize the data transfer of this task with the caller task** is not selected. By default, the block gets the timer value by calling the `tickGet` function in the RTOS (VxWorks). The default resolution is 1/60 second.

`Timer size` — Number of bits to store clock tick
`32bits` (default) | `16bits` | `8bits` | `auto`

The number of bits to store the clock tick for a hardware timer. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the code generator determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. When **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, it uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters”. See also “Timers in Asynchronous Tasks”.

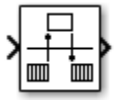
See Also
Async Interrupt

Topics
“Asynchronous Events”

Introduced in R2006a

Unprotected RT

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.

Introduced in R2006a

Code Generation Parameters: Code Generation

Model Configuration Parameters: Code Generation

The **Code Generation** category includes parameters for defining the code generation process including target selection. It also includes parameters for inserting comments and pragmas into the generated code for data and functions. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

These configuration parameters appear in the **Configuration Parameters > Code Generation** general category.

Parameter	Description
“System target file” on page 4-7	Specify which target file configuration will be used.
“Browse” on page 4-9	Browse file configuration options.
“Language” on page 4-10	Specify C or C++ code generation.
“Description” on page 4-12	A description of the target file.
“Generate code only” on page 4-40	Specify code generation versus an executable build.
“Package code and artifacts” on page 4-42	Specify whether to automatically package generated code and artifacts for relocation.
“Zip file name” on page 4-44	Specify the name of the `.zip` file in which to package generated code and artifacts for relocation.
“Compiler optimization level” on page 4-20	Control compiler optimizations for building generated code.
“Custom compiler optimization flags” on page 4-22	Specify custom compiler optimization flags.
“Toolchain” on page 4-13	Specify the toolchain to use when building an executable or library.
“Build configuration” on page 4-15	Specify compiler optimization or debug settings for toolchain.
“Tool/Options” on page 4-18	Display or customize build configuration settings.

Parameter	Description
“Generate makefile” on page 4-24	Enable generation of a makefile based on a template makefile.
“Make command” on page 4-26	Specify a make command and optionally append makefile options.
“Template makefile” on page 4-28	Specify the template makefile from which to generate the makefile.
“Select objective / Prioritized objectives” on page 4-30	Select code generation objectives to use with the Code Generation Advisor.
“Set Objectives” on page 4-33	Open Configuration Set Objectives dialog box.
“Set Objectives — Code Generation Advisor Dialog Box” on page 4-34	Select and prioritize code generation objectives.
“Check model before generating code” on page 4-38	Choose whether to run Code Generation Advisor checks before generating code.
“Check Model” on page 4-37	Check whether the model meets code generation objectives.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Custom FFT library callback” on page 10-67	Specify a callback class for FFTW library calls in code generated for FFT functions in MATLAB code.
“Custom BLAS library callback” on page 10-69	Specify BLAS library callback class for BLAS calls in code generated from MATLAB code.
“Custom LAPACK library callback” on page 10-71	Specify LAPACK library callback class for LAPACK calls in code generated from MATLAB code.
“Verbose build” on page 10-55	Display code generation progress.
“Retain .rtw file” on page 10-57	Specify *`model`*`.rtw` file retention.
“Profile TLC” on page 10-59	Profile the execution time of TLC files.
“Enable TLC assertion” on page 10-65	Produce the TLC stack trace.

Parameter	Description
“Start TLC coverage when generating code” on page 10-63	Generate the TLC execution report.
“Start TLC debugger when generating code” on page 10-61	Specify use of the TLC debugger
“Package” (Embedded Coder)	Specify a package that contains memory sections you want to apply to model-level functions and internal data.
“Refresh package list” (Embedded Coder)	Add user-defined packages that are on the search path to list of packages.
“Initialize/Terminate” (Embedded Coder)	Specify whether to apply a memory section to Initialize/Start and Terminate functions.
“Execution” (Embedded Coder)	Specify whether to apply a memory section to execution functions.
“Shared utility” (Embedded Coder)	Specify whether to apply memory sections to shared utility functions.
“Constants” (Embedded Coder)	Specify whether to apply a memory section to constants.
“Inputs/Outputs” (Embedded Coder)	Specify whether to apply a memory section to root input and output.
“Internal data” (Embedded Coder)	Specify whether to apply a memory section to internal data.
“Parameters” (Embedded Coder)	Specify whether to apply a memory section to parameters.
“Validation results” (Embedded Coder)	Display the results of memory section validation.

The following parameters under **Advanced parameters** are infrequently used and have no other documentation.

Parameter	Description
`PostCodeGenCommand` *`character vector`* `-''`	Add the specified post code generation command to the model build process.

Parameter	Description
`TLCOptions` *`character vector`* - `''`	Specify additional TLC command-line options.

The following parameters are for MathWorks use only.

Parameter	Description
`Comment`	For MathWorks use only.
`PreserveName`	For MathWorks use only.
`PreserveNameWithParent`	For MathWorks use only.
`SignalNamingFcn`	For MathWorks use only.
`TargetTypeEmulationWarn-SuppressLevel` int - **`0`**	For MathWorks use only. When greater than or equal to 2, suppress warning messages that the code generator displays when emulating integer sizes in rapid prototyping environments.

The Configuration Parameters dialog box also includes other code generation parameters:

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Model Configuration Parameters: Code Generation Interface” on page 9-2

See Also

More About

- “Model Configuration”
- “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder)

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**.

To get help on an option

1. Right-click the option's text label.
2. Select **What's This** from the popup menu.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2

System target file

Description

Specify the system target file.

Category: Code Generation

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.

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.
- When you switch from a system target file that is not ERT-based to a file that is ERT-based, the configuration parameter **Default parameter behavior** sets to `Inlined` by default. However, you can change the setting of **Default parameter behavior** later. For more information, see “Default parameter behavior” on page 15-18.
- To configure your model for rapid simulation, select `rsim.tlc`.
- To configure your model for Simulink Real-Time™, select `slrt.tlc`.

Command-Line Information

Parameter: `SystemTargetFile`
Type: character vector
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 ERT based (requires Embedded Coder license)

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Compare System Target File Support Across Products”

Browse

Description

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 4-7**.

Category: Code Generation

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`.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure a System Target File”
- “Compare System Target File Support Across Products”

Language

Description

Specify C or C++ code generation.

Category: 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: character vector

Value: `'C' | 'C++'`
Default: `'C'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Select C or C++ Programming Language”
- “Select and Configure C or C++ Compiler or IDE”
- “Customize Generated C Function Interfaces” (Embedded Coder)
- “Customize Generated C++ Class Interfaces” (Embedded Coder)

Description

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.

Category: Code Generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Browse” on page 4-9

Toolchain

Description

Specify the toolchain to use when building an executable or library.

Note This parameter only appears when the model is configured to use a toolchain-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

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 code generator:

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 the **Configuration Parameters > Code Generation > Advanced parameters > Toolchain > Validate Toolchain** button 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 and build tool in the validation process. If the tool chain definition omits a build tool, validation skips the unspecified tool. To fix a failure (for example, the build tool definition omits a required build tool such as compiler or linker), edit the toolchain definition and repeat the registration process.

Command-Line Information

Parameter: `Toolchain`
Type: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Toolchain Configuration”
- “Adding a Custom Toolchain” (MATLAB Coder)

Build configuration

Description

Specify compiler optimization or debug settings for toolchain.

Note This parameter only appears when the model is configured to use a toolchain-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

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.

This interaction helps synchronize the **Toolchain** value and manually specified **Build configuration** values.

Modifying the **Build configuration** value can affect the **Toolchain** value. The `Automatically locate an installed toolchain` is the only value for **Toolchain** that is affected by changing the **Build configuration** to `Specify`.

- Changing the **Build configuration** from any value to `Specify`, changes the **Toolchain** value `Automatically locate an installed toolchain` (default) to

the value of the toolchain that was located (for example, `Microsoft Visual C++ 2012 v11.0 |(64-bit Windows)`).

- Changing the **Build configuration** from `Specify` to any other value has no effect on the **Toolchain** value.

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.

To extract macro definitions (including compiler optimization flags) from the generated makefile for toolchain approach builds on Windows or UNIX systems, see the *`model`*`.bat` description in “Manage Build Process Files”.

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: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Toolchain Configuration”
- “Adding a Custom Toolchain” (MATLAB Coder)

Tool/Options

Description

Display or customize build configuration settings.

Note These parameters only appear when the model is configured to use the toolchain approach, as described in “Choose Build Approach and Configure Build Process”

Category: Code Generation

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: character vector
Value: Specify the baseline toolchain settings. Use a new-line-delineated character vector to specify each option and its values.
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Toolchain Configuration”
- “Adding a Custom Toolchain” (MATLAB Coder)

Compiler optimization level

Description

Control compiler optimizations for building generated code, using flexible, generalized controls.

Note This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `Optimizations off (faster builds)`

`Optimizations off (faster builds)`

Customizes compilation during the build process to minimize compilation time.

`Optimizations on (faster runs)`

Customizes compilation during the makefile build process to minimize run time.

`Custom`

Allows you to specify custom compiler optimization flags to be applied during the 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 make commands.

- If you specify compiler options for your 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: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Custom compiler optimization flags” on page 4-22
- “Control Compiler Optimizations”

Custom compiler optimization flags

Description

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 a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

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: character vector
Value: `''` | user-specified flags
Default: `''`

Recommended Settings

See “Compiler optimization level” on page 4-20.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Compiler optimization level” on page 4-20
- “Control Compiler Optimizations”

Generate makefile

Description

Enable generation of a makefile based on a template makefile.

Note This option only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Customize Post-Code-Generation Build Processing”
- “Customize Build Process with STF_make_rtw_hook File”
- “Target Development and the Build Process”

Make command

Description

Specify a make command and optionally append makefile options.

Note This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

Settings

Default: `make_rtw`

An internal MATLAB command used by code generation software to control the build process. The specified make command is invoked when you start a build.

- 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 supply makefile options in the **Make command** field. The options are passed to the command-line invocation of the `make` utility, which adds them to the overall flags passed to the compiler. Append the options after the make command, as in the following example:

  ```
  make_rtw OPTS="-DMYDEFINE=1"
  ```

  The syntax for makefile options differs slightly for different compilers.

Tip

- Most targets use the default command.
- You should not invoke `make_rtw` or other internal make commands directly from MATLAB code. To initiate a model build from MATLAB code, use documented build commands such as `slbuild` or `rtwbuild`.

Dependency

This parameter is enabled by **Generate makefile**.

Command-Line Information

Parameter: `MakeCommand`
Type: character vector
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	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Template Makefiles and Make Options”
- “Customize Build Process with STF_make_rtw_hook File”
- “Target Development and the Build Process”

Template makefile

Description

Specify the template makefile from which to generate the makefile.

Note This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in “Choose Build Approach and Configure Build Process”.

Category: Code Generation

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: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Template Makefiles and Make Options”
- “Compare System Target File Support Across Products”

Select objective / Prioritized objectives

Description

Select code generation objectives for reviewing your model configuration settings with the Code Generation Advisor.

Category: Code Generation

Settings

Default: `Unspecified`

`Unspecified`

No objective specified. Do not optimize code generation settings using the Code Generation Advisor. This option only appears for GRT-based targets.

`Debugging`

Specifies debugging objective. Optimize code generation settings for debugging the code generation build process using the Code Generation Advisor. This option only appears for GRT-based targets.

`Execution efficiency`

Specifies execution efficiency objective. Optimize code generation settings to achieve fast execution time using the Code Generation Advisor. This option only appears for GRT-based targets.

Tip

The parameter name displayed for GRT-based targets is **Select objective** and for ERT-based targets, it is **Prioritized objectives**. To configure the code generation objectives with an ERT-based target, the **Prioritized objectives** parameter has an associated **Set Objectives** button. Click the **Set Objectives** button to open the Set Objectives - Code Generation Advisor dialog box.

Dependency

The **Prioritized objectives** parameter and the **Set Objectives** button require Embedded Coder.

Command-Line Information

Parameter: `'ObjectivePriorities'`
Type: cell array of character vectors or string array
Value: `{''}` | `{'Debugging'}` | `{'Execution efficiency'}`
Default: `{''}`

Recommended Settings

Application	Setting
Debugging	`Debugging`
Traceability	Not applicable for GRT-based targets
Efficiency	`Execution efficiency`
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Prioritized objectives

Description

List objectives that you specify by clicking the **Set Objectives** button.

Category: Code Generation

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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Set Objectives

Description

Open Configuration Set Objectives dialog box.

Category: Code Generation

Dependency

This button appears only for ERT-based targets.

See Also

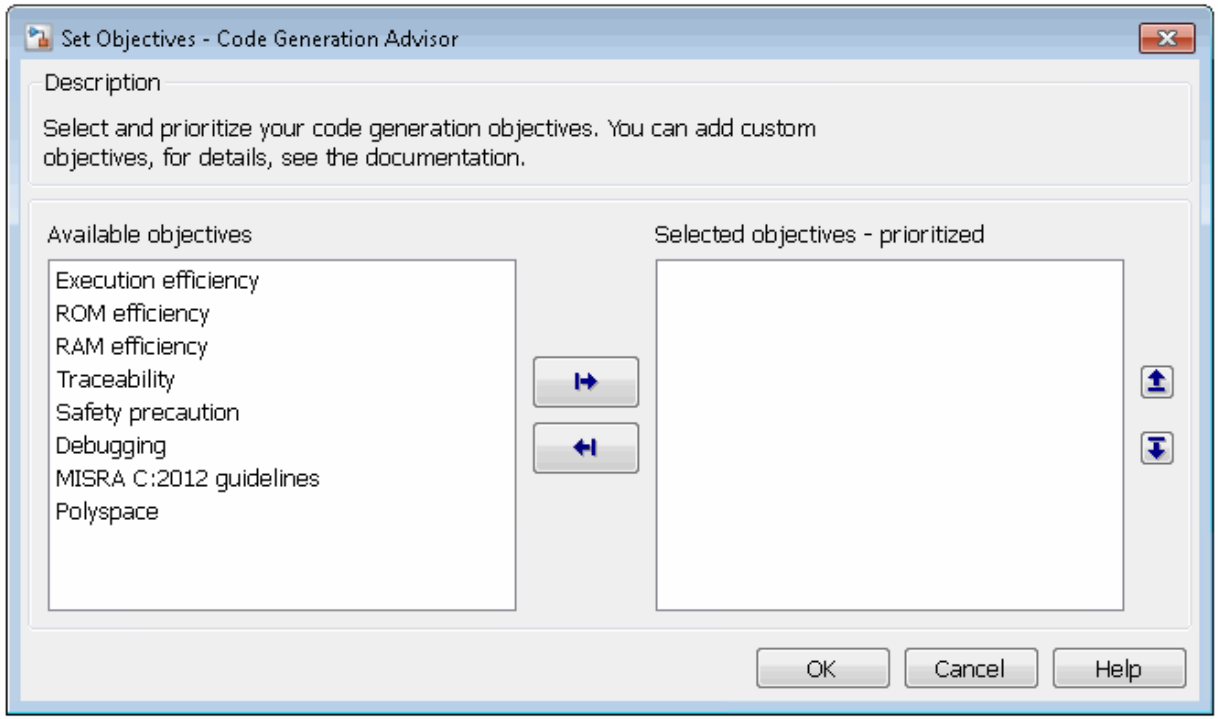
Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Set Objectives — Code Generation Advisor Dialog Box

Description

Select and prioritize code generation objectives to use with the Code Generation Advisor.



Category: Code Generation

Settings

1. From the **Available objectives** list, select objectives.
2. Click the select button (arrow pointing right) to move the objectives that you selected into the **Selected objectives - prioritized** list.
3. Click the higher priority (up arrow) and lower priority (down arrow) buttons to prioritize the objectives.

Objectives

List of available objectives.
`Execution efficiency` — Configure code generation settings to achieve fast execution time.
`ROM efficiency` — Configure code generation settings to reduce ROM usage.
`RAM efficiency` — Configure code generation settings to reduce RAM usage.
`Traceability` — Configure code generation settings to provide mapping between model elements and code.
`Safety precaution` — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
`Debugging` — Configure code generation settings to debug the code generation build process.
`MISRA C:2012 guidelines` — Configure code generation settings to increase compliance with MISRA C:2012 guidelines.
`Polyspace` — Configure code generation settings to prepare the code for Polyspace® analysis.

Note If you select the MISRA C:2012 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA C:2012 configuration setting recommendations.
- For blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Priorities

After you select objectives from the **Available objectives** parameter, organize the objectives in the **Selected objectives - prioritized** parameter with the highest priority objective at the top.

Dependency

This dialog box appears only for ERT-based targets.

Command-Line Information

Parameter: `'ObjectivePriorities'`
Type: cell array of character vectors or string array; combination of the available values
Value: `{''}|{'Execution efficiency'}|{'ROM efficiency'}|{'RAM efficiency'}|{'Traceability'}|{'Safety precaution'}|{'Debugging'}|{'MISRA C:2012 guidelines'}|{'Polyspace'}`
Default: `{''}`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Check Model

Description

Run the Code Generation Advisor checks.

Category: Code Generation

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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Check model before generating code

Description

Choose whether to run Code Generation Advisor checks before generating code.

Category: Code Generation

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 code generator continues producing 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 code generator does not continue producing code.

Command-Line Information

Parameter: `CheckMdlBeforeBuild`
Type: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”

Generate code only

Description

Specify code generation versus an executable build.

Category: Code Generation

Settings

Default: off

 On

The build process generates code and a makefile, but it does not invoke the make command.

 Off

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**.

Command-Line Information

Parameter: `GenCodeOnly`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Customize Post-Code-Generation Build Processing”

Package code and artifacts

Description

Specify whether to automatically package generated code and artifacts for relocation.

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Relocate Code to Another Development Environment”
- “packNGo Function Limitations”

Zip file name

Description

Specify the name of the `.zip` file in which to package generated code and artifacts for relocation.

Category: Code Generation

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: character vector
Value: valid name for a `.zip` file
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Relocate Code to Another Development Environment”
- “packNGo Function Limitations”

Code Generation Parameters: Report

Model Configuration Parameters: Code Generation Report

The **Code Generation > Report** category includes parameters for generating and customizing the code generation report. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Report** pane.

Parameter	Description
“Create code generation report” on page 5-5	Document generated code in an HTML report.
“Open report automatically” on page 5-8	Specify whether to display code generation reports automatically.
“Generate model Web view” on page 5-10	Include the model Web view in the code generation report to navigate between the code and model within the same window.
“Static code metrics” on page 5-12	Include static code metrics report in the code generation report.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Code-to-model” on page 10-6	Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.
“Model-to-code” on page 10-8	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.

Parameter	Description
“Configure” on page 10-10	Open the **Model-to-code navigation** dialog box for specifying a build folder containing previously-generated model code to highlight.
“Eliminated / virtual blocks” on page 10-11	Include summary of eliminated and virtual blocks in code generation report.
“Traceable Simulink blocks” on page 10-13	Include summary of Simulink blocks in code generation report.
“Traceable Stateflow objects” on page 10-15	Include summary of Stateflow objects in code generation report.
“Traceable MATLAB functions” on page 10-17	Include summary of MATLAB functions in code generation report.
“Summarize which blocks triggered code replacements” on page 10-19	Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

See Also

More About

- “Report Generation”
- “Model Configuration”

Code Generation: Report Tab Overview

Control the code generation report that the code generator automatically creates.

Configuration

To create a code generation report during the build process, select the **Create code generation report** parameter.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Generate a Code Generation Report”
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions” (Embedded Coder)

Create code generation report

Description

Document generated code in an HTML report.

Category: Code Generation > Report

Settings

Default: On

 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 system target file).
- 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 system target file).
- 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 system target file).
- 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 system target file).

 Off

Does not generate a summary of files.

Dependency

This parameter enables and selects

- **Open report automatically** on page 5-8
- **Code-to-model** on page 10-6 (ERT target)

This parameter enables

- **Model-to-code** on page 10-8 (ERT target)
- **Eliminated / virtual blocks** on page 10-11 (ERT target)
- **Traceable Simulink blocks** on page 10-13 (ERT target)
- **Traceable Stateflow objects** on page 10-15 (ERT target)
- **Traceable MATLAB functions** on page 10-17 (ERT target)

.

Command-Line Information

Parameter: `GenerateReport`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions” (Embedded Coder)
- “Configure Code Coverage with Third-Party Tools” (Embedded Coder)

Open report automatically

Description

Specify whether to display code generation reports automatically.

Category: Code Generation > Report

Settings

Default: On

 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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions” (Embedded Coder)

Generate model Web view

Description

Include the model Web view in the code generation report to navigate between the code and model within the same window. You can share your model and generated code outside of the MATLAB environment. You must have a Simulink Report Generator™ license to include a Web view (Simulink Report Generator) of the model in the code generation report.

Category: 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.

Dependencies

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**.
- To enable traceability between the code and model, select **Code-to-model** and **Model-to-code**.

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

“Model Configuration Parameters: Code Generation Report” on page 5-2

Related Examples

- “Web View of Model in Code Generation Report” (Embedded Coder)

Static code metrics

Description

Include static code metrics report in the code generation report.

Category: Code Generation > Report

Settings

Default: Off

 On

Include static code metrics report in the code generation report.

 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

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Static Code Metrics” (Embedded Coder)

Code Generation Parameters: Comments

Model Configuration Parameters: Code Generation Comments

The **Code Generation > Comments** category includes parameters for configuring the comments in the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Comments** pane.

Parameter	Description
“Include comments” on page 6-5	Specify which comments are in generated files.
“Simulink block comments” on page 6-7	Specify whether to insert Simulink block comments.
“Trace to model using” on page 6-9	Specify format of comments for Simulink blocks, Stateflow elements and MATLAB function blocks.
“Stateflow object comments” on page 6-11	Specify whether to insert Stateflow object comments.
“MATLAB source code as comments” on page 6-13	Specify whether to insert MATLAB source code as comments.
“Show eliminated blocks” on page 6-15	Specify whether to insert eliminated block's comments.
“Verbose comments for 'Model default' storage class” on page 6-17	Reduce code size or improve code traceability by controlling the generation of comments.
“Operator annotations” on page 6-19	Specify whether to include operator annotations for Polyspace in the generated code as comments.
“Simulink block descriptions” on page 6-21	Specify whether to insert descriptions of blocks into generated code as comments.

Parameter	Description
“Stateflow object descriptions” on page 6-29	Specify whether to insert descriptions of Stateflow objects into generated code as comments.
“Simulink data object descriptions” on page 6-23	Specify whether to insert descriptions of Simulink data objects into generated code as comments.
“Requirements in block comments” on page 6-31	Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.
“Custom comments (MPT objects only)” on page 6-25	Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.
“MATLAB user comments” on page 6-33	Specify whether to include MATLAB user comments as comments.
“Custom comments function” on page 6-27	Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects.

The following configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Comment style” on page 10-83	Specify a multi-line or single-line comment style for generated C or C++ code.

See Also

More About

- “Configure Code Comments”
- “Verify Generated Code by Using Code Tracing” (Embedded Coder)

Code Generation: Comments Tab Overview

Control the comments that the code generator creates and inserts into the generated code.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2

Include comments

Description

Specify which comments are in generated files.

Category: Code Generation > Comments

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 comments** on page 6-7
- **Stateflow object comments** on page 6-11
- **MATLAB source code as comments** on page 6-13
- **Show eliminated blocks** on page 6-15
- **Verbose comments for 'Model default' storage class** on page 6-17

Command-Line Information

Parameter: `GenerateComments`
Type: character vector

Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Configure Code Comments”
- “Verify Generated Code by Using Code Tracing” (Embedded Coder)

Simulink block comments

Description

Specify whether to insert Simulink block comments.

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts automatically generated comments that describe a block's code. The comments precede generated code in the generated file.

 Off

Suppresses comments.

Dependency

- **Include comments** on page 6-5 enables this parameter.
- This parameter enables **Trace to model using** on page 6-9.

Command-Line Information

Parameter: `SimulinkBlockComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Trace Simulink Model Elements in Generated Code” (Embedded Coder)

Trace to model using

Description

Specify format of comments for Simulink blocks, Stateflow elements, and MATLAB function blocks.

Category: Code Generation > Comments

Settings

Default: `Block path`

Block path

The generated comment includes the entire block path from the root as the traceability link. For example:

```
/* Outport: '<Root>/Out1' */
rtwdemo_comments_Y.Out1 = 1;
```

Simulink identifier

The generated comment includes the Simulink identifier without the model name for the corresponding block or object. For example:

```
/* Outport: 'Out1' (':33') */
rtwdemo_comments_Y.Out1 = 1;
```

Dependency

- This parameter requires Embedded Coder.
- **Simulink block comments** on page 6-7 or **Stateflow object comments** on page 6-11 enable this parameter.

Command-Line Information

Parameter: `BlockCommentType`
Type: character vector
Value: `'Block path'` | `'Simulink identifier'`

Default: `'Block path'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Locate Diagram Components Using Simulink Identifiers” (Simulink)

Stateflow object comments

Description

Specify whether to insert Stateflow object comments.

Category: Code Generation > Comments

Settings

Default: Off

 On

Inserts automatically generated comments that contain Stateflow object IDs or MATLAB code line locations. The comments precede the generated code in the generated file. For example,

```
/* Entry 'First': '<S2>:2' */
rtY.Out1 = 1;
```

`'<S2>:2'` is a hyperlinked traceability tag that facilitates tracing of generated code to corresponding Stateflow element.

 Off

Suppresses comments.

Dependency

- **Include comments** on page 6-5 enables this parameter.
- This parameter enables **Trace to model using** on page 6-9.

Command-Line Information

Parameter: `StateflowObjectComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Trace Stateflow Elements in Generated Code” (Embedded Coder)

MATLAB source code as comments

Description

Specify whether to insert MATLAB source code as comments.

Category: Code Generation > Comments

Settings

Default: Off

 On

Inserts MATLAB source code as comments in the generated code. The comment appears after the traceability tag and precedes the associated generated code. For example,

```
/* '<S2>:1:22' xb1 = x-1; */
xb1 = x;
```

Selecting this parameter adds the MATLAB code `xb1 = x-1;` in the traceability comment.

Includes the function signature in the function banner.

 Off

Suppresses comments and does not include the function signature in the function banner.

Dependency

Include comments on page 6-5 enables this parameter.

Command-Line Information

Parameter: `MATLABSourceComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Include MATLAB Code as Comments in Generated Code”

Show eliminated blocks

Description

Specify whether to insert eliminated block's comments.

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts statements in the generated code from blocks eliminated as the result of optimizations (such as parameter inlining).

 Off

Suppresses statements.

Dependency

Include comments on page 6-5 enables this parameter.

Command-Line Information

Parameter: `ShowEliminatedStatement`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2

Verbose comments for 'Model default' storage class

Description

Reduce code size or improve code traceability by controlling the generation of comments. The comments appear interleaved in the code that initializes the fields of the model parameter structure, which appears in the *`model`*`_data.c` file or the *`model`*`.c` file. Each comment indicates the name of a parameter object (`Simulink.Parameter`) or MATLAB variable and the blocks that use the object or variable to set parameter values.

Parameter objects and MATLAB variables appear in the model parameter structure under either of these conditions:

- You apply the storage class `Model default` to the object or variable and, in the Code Mapping Editor, you set the storage class of the corresponding category of data to the default setting, `Default`.
- You apply the storage class `Auto` to the object or variable and set the model configuration parameter **Default parameter behavior** to `Tunable`. In the Code Mapping Editor, you set the storage class of the corresponding category of data to the default setting, `Default`.

For more information about parameter representation in the generated code, see “How Generated Code Stores Internal Signal, State, and Parameter Data”.

Category: Code Generation > Comments

Settings

Default: On

 On

Generate comments regardless of the number of parameter values stored in the parameter structure. Use this setting to improve traceability between the generated code and the parameter objects or variables that the model uses.

 Off

Generate comments only if the parameter structure contains fewer than 1000 parameter values. An array parameter with *n* elements represents *n* values. For large models, use this setting to reduce the size of the generated file.

Dependency

Include comments on page 6-5 enables this parameter.

Command-Line Information

Parameter: `ForceParamTrailComments`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Operator annotations

Description

Specify whether to include operator annotations for Polyspace in the generated code as comments.

Category: Code Generation > Comments

Settings

Default: On

 On

Includes operator annotations in the generated code.

 Off

Does not include operator annotations in the generated code.

Tips

- These annotations help document overflow behavior that is due to the way the code generator implements an operation. These operators cannot be traced to an overflow in the design.
- Justify operators that the Polyspace software cannot prove. When this option is enabled, if the code generator uses one of these operators, it adds annotations to the generated code to justify the operators for Polyspace.
- The code generator cannot justify operators that result from the design.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Include comments** on page 6-5 enables this parameter.

Command-Line Information

Parameter: `OperatorAnnotations`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Annotate Code for Justifying Polyspace Checks” (Embedded Coder)

Simulink block descriptions

Description

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

Category: Code Generation > Comments

Settings

Default: On

 On

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

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

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

 Off

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

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Internationalization and Code Generation”

Simulink data object descriptions

Description

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

This parameter does not affect `Simulink.LookupTable` or `Simulink.Breakpoint` objects that you configure to appear in the generated code as a structure (for example, by storing all of the table and breakpoint data in a single `Simulink.LookupTable` object).

Category: Code Generation > Comments

Settings

Default: On

 On

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

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

 Off

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

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: character vector

Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2

Custom comments (MPT objects only)

Description

Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code. MPT data objects are objects of the classes `mpt.Parameter` and `mpt.Signal`.

Category: Code Generation > Comments

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Add Custom Comments for Variables in the Generated Code” (Embedded Coder)

Custom comments function

Description

Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects. MPT data objects are objects of the classes `mpt.Parameter` and `mpt.Signal`.

Category: Code Generation > Comments

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.

For an example MATLAB function, see the function *`matlabroot`*`/toolbox/rtw/rtwdemos/rtwdemo_comments_mptfun.m`.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Custom comments (MPT objects only)** enables this parameter.

Command-Line Information

Parameter: `CustomCommentsFcn`
Type: character vector
Value: valid file name
Default: `''`

Recommended Settings

Application	Setting
Debugging	Valid file name
Traceability	Valid file name
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Add Custom Comments for Variables in the Generated Code” (Embedded Coder)

Stateflow object descriptions

Description

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

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts descriptions of Stateflow states, charts, transitions, and graphical functions into generated code as comments. The descriptions come from the **Description** field in Object Properties pane in the Model Explorer for these Stateflow objects. The comments appear just above the code generated for each object.

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

 Off

Suppresses the generation of comments for Stateflow objects.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Stateflow license.

Command-Line Information

Parameter: `SFDataObjDesc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	Noimpact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Internationalization and Code Generation”

Requirements in block comments

Description

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

Category: Code Generation > Comments

Settings

Default: Off

 On

Inserts the requirement descriptions that you assign to Simulink blocks into the generated code as comments. The code generator includes the requirement descriptions in the generated code in the following locations.

Model Element	Requirement Description Location
Model	In the main header file *`model`*`.h`
Nonvirtual subsystems	At the call site for the subsystem
Virtual subsystems	At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem does not have a nonvirtual parent, requirement descriptions are located in the main header file for the model, *`model`*`.h`.
Nonsubsystem blocks	In the generated code for the block

For information on code generator treatment of strings that are unrepresented in the character set encoding for the model, see “Internationalization and Code Generation”.

 Off

Suppresses the generation of comments for block requirement descriptions.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires Embedded Coder and Simulink Check™ licenses when generating code.

Tips

If you use an external `.req` file to store your requirement links, to avoid stale comments in generated code, before code generation, you must save any change in your requirement links.

Command-Line Information

Parameter: `ReqsInCode`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “How Requirements Information Is Included in Generated Code” (Simulink Requirements)

MATLAB user comments

Description

Specify whether to include MATLAB user comments including both function description comments and other user comments from MATLAB code as comments in the generated code.

Category: Code Generation > Comments

Settings

Default: Off

 On

Inserts MATLAB user comments as comments.

 Off

Suppresses comments.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- **Include comments** on page 6-5 enables this parameter.

Command-Line Information

Parameter: `MATLABFcnDesc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Noimpact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Include MATLAB Code as Comments in Generated Code”

Code Generation Parameters: Symbols

Model Configuration Parameters: Code Generation Symbols

The **Code Generation > Symbols** category includes parameters for configuring the comments in the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Symbols**.

Parameter	Description
“Global variables” on page 7-6	Customize generated global variable identifiers.
“Global types” on page 7-9	Customize generated global type identifiers.
“Field name of global types” on page 7-12	Customize generated field names of global types.
“Subsystem methods” on page 7-15	Customize generated function names for reusable subsystems.
“Subsystem method arguments” on page 7-18	Customize generated function argument names for reusable subsystems.
“Local temporary variables” on page 7-20	Customize generated local temporary variable identifiers.
“Local block output variables” on page 7-23	Customize generated local block output variable identifiers.
“Constant macros” on page 7-25	Customize generated constant macro identifiers.
“Shared utilities identifier format” on page 7-28	Customize shared utility identifiers.
“Minimum mangle length” on page 7-31	Specify the minimum number of characters for generating name-mangling text to help avoid name collisions.

Parameter	Description
“Maximum identifier length” on page 7-33	Specify maximum number of characters in generated function, type definition, variable names.
“System-generated identifiers” on page 7-35	Specify whether the code generator uses shorter, more consistent names for the $N token in system-generated identifiers.
“Generate scalar inlined parameters as” on page 7-40	Control expression of scalar inlined parameter values in the generated code.
“Use the same reserved names as Simulation Target” on page 7-56	Specify whether to use the same reserved names as those specified in the **Simulation Target** pane.
“Reserved names” on page 7-58	Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

The following configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Shared checksum length” on page 10-73	Specify character length of `$C` token.
“EMX array utility functions identifier format” on page 10-75	Customize generated identifiers for `emxArray` (embeddable `mxArray`) utility functions.
“EMX array types identifier format” on page 10-78	Customize generated identifiers for `emxArray` (embeddable `mxArray`) types.
“Custom token text” on page 7-60	Specify text to insert for $U token.
“Signal naming” on page 7-44	Specify rules for naming signals in generated code.
“M-function” on page 7-46	
“Parameter naming” on page 7-48	Specify rule for naming parameters in generated code.
“M-function” on page 7-50	

Parameter	Description
“#define naming” on page 7-52	Specify rule for naming `#define` parameters (defined with storage class `Define (Custom)`) in generated code.
“M-function” on page 7-54	

See Also

More About

- “Code Appearance”
- “Model Configuration”

Code Generation: Symbols Tab Overview

Select the automatically generated identifier naming rules.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “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” (Embedded Coder)

Global variables

Description

Customize generated global variable identifiers.

Category: Code Generation > Symbols

Settings

Default: `$R$N$M`

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

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.
`$G`	Insert the name of a storage class that is associated with the data item.

Tips

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

- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This parameter setting only determines the name of objects, such as signals and parameters, if the object is set to `Auto`.
- For referenced models, if the **Global variables** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

  You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

  1. In the Simulink Editor, select **Analysis** > **Model Advisor**.
  2. Select **By Task**.
  3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrGlobalVar`
Type: character vector
Value: valid combination of tokens
Default: `$R$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Avoid Identifier Name Collisions with Referenced Models” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Global types

Description

Customize generated global type identifiers.

Category: Code Generation > Symbols

Settings

Default: `$N$R$M_T`

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

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.
`$G`	Insert the name of a storage class that is associated with the data item.

Tips

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

- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For referenced models, if the **Global types** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

  You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

  1. In the Simulink Editor, select **Analysis > Model Advisor**.
  2. Select **By Task**.
  3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrType`
Type: character vector

Value: valid combination of tokens
Default: `$N$R$M_T`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Avoid Identifier Name Collisions with Referenced Models” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Field name of global types

Description

Customize generated field names of global types.

Category: Code Generation > Symbols

Settings

Default: `$N$M`

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

Token	Description
`$A`	Insert data type acronym into signal and work vector identifiers. For example, `i32` for `int32_t`.
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the text `root_`. For blocks at the subsystem level, the tag is of the form `sN_`, where `N` is a unique system number assigned by the Simulink software.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).
- 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.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrField`
Type: character vector
Value: valid combination of tokens
Default: `$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Subsystem methods

Description

Customize generated function names for reusable subsystems.

Category: Code Generation > Symbols

Settings

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

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

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

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For referenced models, if the **Subsystem methods** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

  You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

  1. In the Simulink Editor, select **Analysis > Model Advisor**.
  2. Select **By Task**.
  3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrFcn`
Type: character vector
Value: valid combination of tokens
Default: `$R$N$M$F`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Avoid Identifier Name Collisions with Referenced Models” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Subsystem method arguments

Description

Customize generated function argument names for reusable subsystems.

Category: Code Generation > Symbols

Settings

Enter a macro that specifies whether, and in what order, certain text is to be included in the generated argument name. The macro can include a combination of the following format tokens.

Token	Description
`$I`	• Insert `u` if the argument is an input. • Insert `y` if the argument is an output. • Insert `uy` if the argument is an input and output. Optional.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. Recommended to maximize readability of generated code.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

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

- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrFcnArg`
Type: character vector
Value: valid combination of tokens
Default: `rt$I$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Local temporary variables

Description

Customize generated local temporary variable identifiers.

Category: Code Generation > Symbols

Settings

Default: `$N$M`

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

Token	Description
`$A`	Insert data type acronym (for example, `i32` for integers) into signal and work vector identifiers.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).
- 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.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrTmpVar`
Type: character vector
Value: valid combination of tokens
Default: `$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Avoid Identifier Name Collisions with Referenced Models” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Local block output variables

Description

Customize generated local block output variable identifiers.

Category: Code Generation > Symbols

Settings

Default: `rtb_$N$M`

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

Token	Description
`$A`	Insert data type acronym (for example, `i32` for integers) into signal and work vector identifiers.
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).

- 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.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrBlkIO`
Type: character vector
Value: valid combination of tokens
Default: `rtb_$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Constant macros

Description

Customize generated constant macro identifiers.

Category: Code Generation > Symbols

Settings

Default: `$R$N$M`

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

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

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

- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).
- For referenced models, if the **Constant macros** parameter does not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

  You can use the Model Advisor to identify models in a model referencing hierarchy for which code generation changes configuration parameter settings.

  1. In the Simulink Editor, select **Analysis** > **Model Advisor**.
  2. Select **By Task**.
  3. Run the **Check code generation identifier formats used for model reference** check.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrMacro`
Type: character vector
Value: valid combination of tokens
Default: `$R$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Avoid Identifier Name Collisions with Referenced Models” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Shared utilities identifier format

Description

Customize shared utility identifiers.

Category: Code Generation > Symbols

Settings

Default: `$N$C`

Customize generated shared utility identifier names.

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

Token	Description
`$N`	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated. Optional.
`$C`	Insert eight-character conditional checksum when `$N` is not specified or the**Maximum identifier length** does not accommodate the full length of `$N`. Modify checksum character length by using **Shared checksum length** parameter. Required.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character.
`$U`	Insert text that you specify for the `$U` token. Use the **Custom token text** parameter to specify this text.

Tips

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate.
- The checksum token `$C` is required. If `$C` is specified without `$N` or `$R`, the checksum is included in the identifier name. Otherwise, the code generator includes the checksum when necessary to prevent name collisions.

- To control the case (upper or lower case) of the text that each token represents, include decorators such as `[U_]` in your macro. See “Control Case with Token Decorators” (Embedded Coder).
- If you specify `$N` or `$R`, then the checksum is only included in the name when the identifier length is too short to accommodate the fully expanded format text. The code generator includes the checksum and truncates `$N` or `$R` until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.
- If you specify `$N` and `$R`, then the checksum is only included in the name when the identifier length is too short to accommodate the fully expanded format text. The code generator includes the checksum and truncates `$N` until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.
- Descriptive text helps make the identifier name more accessible.
- For versions prior to R2016a, the **Shared utilities identifier format** parameter does not support the `$R` token. For a model, if the **Shared utilities identifier format** parameter includes a `$R` token, and you export the model to a version prior to R2016a, the **Shared utilities identifier format** parameter defaults to `$N$C`.

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrUtil`
Type: character vector
Value: valid combination of tokens
Default: `$N$C`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Exceptions to Identifier Formatting Conventions” (Embedded Coder)

Minimum mangle length

Description

Increase the minimum number of characters for generating name-mangling text to help avoid name collisions.

Category: Code Generation > Symbols

Settings

Default: 1

Specify an integer value that indicates the minimum number of characters the code generator uses when generating name-mangling text. The maximum possible value is 15. The minimum value automatically increases during code generation as a function of the number of collisions. A larger value reduces the chance of identifier disturbance when you modify the model.

Tips

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

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Maintain Traceability for Generated Identifiers” (Embedded Coder)

Maximum identifier length

Description

Specify maximum number of characters in generated function, type definition, variable names.

Category: Code Generation > Symbols

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 text. 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Identifier Format Control” (Embedded Coder)

System-generated identifiers

Description

Specify whether the code generator uses shorter, more consistent names for the `$N` token in system-generated identifiers.

Category: Code Generation > Symbols

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 7-6 is `$N$R$M`, the block state identifier is `sym_DWork`.
- "Global types" on page 7-9 is `$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 7-6 is `$N$R$M`, the block state identifier is `sym_DW`.
- "Global types" on page 7-9 is `$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, Global Variable	Block signals of the system
`ExternalInputs`	`ExtU`	Type	Block input data for root system
`ExternalInputSizes`	`ExtUSize`	Type	Size of block input data for the root system (used when inputs are variable dimensions)
`ExternalOutputs`	`ExtY`	Type	Block output data for the root system
`ExternalOutputSizes`	`ExtYSize`	Type	Size of block output data for the root system
`U`	`U`	Global Variable	Input data
`USize`	`USize`	Global Variable	Size of input data
`Y`	`Y`	Global Variable	Output data
`YSize`	`YSize`	Global Variable	Size of output data
`Parameters`	`P`	Type, Global Variable	Parameters for the system
`ConstBlockIO`	`ConstB`	`Const` Type, Global Variable	Block inputs and outputs that are constants
`MachineLocalData, Machine`	`MachLocal`	`Const` Type, Global Variable	Used by 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

Classic	Shortened	Data Representation	Description
`MassMatrixGlobal`	`MassMatrix`	Type, Global Variable	Used for physical modeling blocks
`PrevZCSigStates`, `PrevZCSigState`	`PrevZCX`	Type, Global Variable	Previous zero-crossing signal state
`ContinuousStates`, `X`	`X`	Type, Global Variable	Continuous states
`StateDisabled`, `Xdis`	`XDis`	Type, Global Variable	Status of an enabled subsystem
`StateDerivatives`, `Xdot`	`XDot`	Type, Global Variable	Derivatives of continuous states at each time step
`ZCSignalValues`, `ZCSignalValue`	`ZCV`	Type, Global Variable	Zero-crossing signals
`DefaultParameters`	`DefaultP`	Global Variable	Default parameters in the system
`GlobalTID`	`GlobalTID`	Global Variable	Used for sample time for states in referenced models
`InvariantSignals`	`Invariant`	Global Variable	Invariant signals
`NSTAGES`	`NSTAGES`	Global Variable	Solver macro
`Object`	`Obj`	Global Variable	Used by ERT C++ code generation to refer to referenced model objects
`TimingBridge`	`TimingBrdg`	Global Variable	Timing information stored in different data structures
`SharedDSM`	`SharedDSM`	Type, Global Variable	Shared local data stores, which are Data Store Memory blocks with **Share across model instances** selected

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.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `InternalIdentifier`

Type: character vector
Value: `Classic` | `Shortened`
Default: `Shortened`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Specify Identifier Length to Avoid Naming Collisions”
- “Specify Reserved Names for Generated Identifiers”
- “Standard Data Structures in the Generated Code”
- “Customize Generated Identifier Naming Rules” (Embedded Coder)
- “Identifier Format Control” (Embedded Coder)

Generate scalar inlined parameters as

Description

Control expression of scalar inlined parameter values in the generated code. Block parameters appear inlined in the generated code when you set **Configuration Parameters** > **Optimization** > **Default parameter behavior** to `Inlined`.

Category: Code Generation > Symbols

Settings

Default: `Literals`

`Literals`

Generates scalar inlined parameters as numeric constants.

`Macros`

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

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `InlinedPrmAccess`
Type: character vector
Value: `Literals` | `Macros`
Default: `Literals`

Recommended Settings

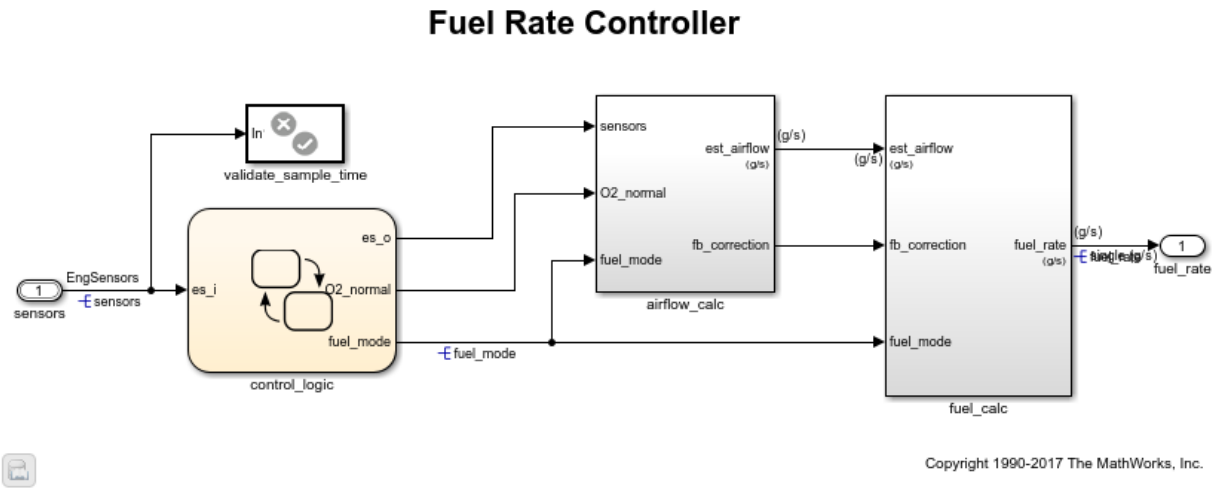
Application	Setting
Debugging	No impact
Traceability	`Macros`
Efficiency	No impact
Safety precaution	No impact

Improve Code Readability by Generating Block Parameter Values as Macros

When you generate efficient code by inlining the numeric values of block parameters (with the configuration parameter **Default parameter behavior**), you can configure scalar parameters to appear as macros instead of literal numbers. Each macro has a unique name that is based on the name of the corresponding block parameter.

Open the example model `sldemo_fuelsys_dd_controller`.

```
sldemo_fuelsys_dd_controller
```



The model uses these configuration parameter settings:

- **Default parameter behavior** set to `Inlined`.
- **System target file** set to `ert.tlc`.

Set the configuration parameter **Generate scalar inlined parameters as** to `Macros`.

```
set_param('sldemo_fuelsys_dd_controller','InlinedPrmAccess','Macros')
```

Generate code from the model.

```
rtwbuild('sldemo_fuelsys_dd_controller')
```

```
### Starting build procedure for model: sldemo_fuelsys_dd_controller
### Successful completion of code generation for model: sldemo_fuelsys_dd_controller
```

The header file `sldemo_fuelsys_dd_controller_private.h` defines several macros that represent inlined (nontunable) block parameters. For example, the macros `rtCP_DiscreteFilter_NumCoe_EL_0` and `rtCP_DiscreteFilter_NumCoe_EL_1` represent floating-point constants.

```
file = fullfile('sldemo_fuelsys_dd_controller_ert_rtw',...
    'sldemo_fuelsys_dd_controller_private.h');
rtwdemodbtype(file,'#define rtCP_DiscreteFilter_NumCoe_EL_0',...
    'rtCP_DiscreteFilter_NumCoe_EL_1',1,1)
```

```
#define rtCP_DiscreteFilter_NumCoe_EL_0 (8.7696F)
#define rtCP_DiscreteFilter_NumCoe_EL_1 (-8.5104F)
```

The comments above the macro definitions indicate that the code generated for a Discrete Filter block uses the macros.

```
rtwdemodbtype(file,'Computed Parameter: DiscreteFilter_NumCoef',...
    'Referenced by: ''<S12>/Discrete Filter''',1,1)
```

```
/* Computed Parameter: DiscreteFilter_NumCoef
 * Referenced by: '<S12>/Discrete Filter'
```

Click the hyperlink to navigate to the block in the model.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2

Signal naming

Description

Specify rules for naming `mpt` signals in generated code.

Category: Code Generation > Symbols

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 a `mpt.Signal` data object, that value overrides the specification of the **Signal naming** parameter.

Limitation

This parameter works only for `mpt.Signal` data objects.

Command-Line Information

Parameter: `SignalNamingRule`
Type: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Apply Naming Rules to Simulink Data Objects” (Embedded Coder)
- “Programming Scripts and Functions” (MATLAB)

M-function

Description

Specify rule for naming identifiers in generated code.

Category: Code Generation > Symbols

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.

For example, the following function returns an identifier name by appending the text `_signal` to a signal data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_signal';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **Signal naming**.
- Must be the same for top-level and referenced models.

Command-Line Information

Parameter: `SignalNamingFcn`
Type: character vector
Value: MATLAB language file
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule Using a Function” (Embedded Coder)
- “Programming Scripts and Functions” (MATLAB)

Parameter naming

Description

Specify rule for naming `mpt` parameters in generated code.

Category: Code Generation > Symbols

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.

Limitation

This parameter works only for `mpt.Parameter` data objects.

Command-Line Information

Parameter: `ParamNamingRule`
Type: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Apply Naming Rules to Simulink Data Objects” (Embedded Coder)
- “Programming Scripts and Functions” (MATLAB)

M-function

Description

Specify rule for naming identifiers in generated code.

Category: Code Generation > Symbols

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.

For example, the following function returns an identifier name by appending the text `_param` to a parameter data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_param';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **Parameter naming**.
- Must be the same for top-level and referenced models.

Command-Line Information

Parameter: `ParamNamingFcn`
Type: character vector
Value: MATLAB language file
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule Using a Function” (Embedded Coder)
- “Programming Scripts and Functions” (MATLAB)

#define naming

Description

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

Category: Code Generation > Symbols

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: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule for Storage Class Define” (Embedded Coder)
- “Programming Scripts and Functions” (MATLAB)

M-function

Description

Specify rule for naming identifiers in generated code.

Category: Code Generation > Symbols

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.

For example, the following function returns an identifier name by appending the text `_define` to a data object name.

```
function revisedName = append_text(name, object)
% APPEND_TEXT: Returns an identifier for generated
% code by appending text to a #define data object name.
%
% Input arguments:
% name: data object name as spelled in model
% object: target data object
%
% Output arguments:
% revisedName: altered identifier returned for use in
% generated code.
%
%
text = '_define';

revisedName = [name,text];
```

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by **#define naming**.
- Must be the same for top-level and referenced models.

Command-Line Information

Parameter: `DefineNamingFcn`
Type: character vector
Value: MATLAB language file
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule Using a Function” (Embedded Coder)
- “Programming Scripts and Functions” (MATLAB)

Use the same reserved names as Simulation Target

Description

Specify whether to use the same reserved names as those specified in the **Simulation Target** pane.

Category: Code Generation > Symbols

Settings

Default: Off

 On

Enables using the same reserved names as those specified in the **Simulation Target** pane.

 Off

Disables using the same reserved names as those specified in the **Simulation Target** pane.

Command-Line Information

Parameter: `UseSimReservedNames`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2

Reserved names

Description

Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

Category: Code Generation > Symbols

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 code generator 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: cell array of character vectors or 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

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2

Custom token text

Description

Specify text to insert for $U token.

Category: Code Generation > Symbols

Settings

Default: `''`

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code

Command-Line Information

Parameter: `CustomUserTokenString`
Type: `character vector`
Value: `''` or `user-specified name`
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Set a custom string and use $U in symbols
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Identifier Format Control” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Symbols” on page 7-2

8

Code Generation Parameters: Custom Code

Model Configuration Parameters: Code Generation Custom Code

The **Code Generation > Custom Code** category includes parameters for inserting custom C code into the generated code. These parameters require a Simulink Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Custom Code** pane.

Parameter	Description
"Use the same custom code settings as Simulation Target" on page 8-5	Specify whether to use the same custom code settings as those in the **Simulation Target > Custom Code** pane.
"Source file" on page 8-7	Specify custom code to include near the top of the generated model source file.
"Header file" on page 8-9	Specify custom code to include near the top of the generated model header file.
"Initialize function" on page 8-11	Specify custom code to include in the generated model initialize function.
"Terminate function" on page 8-13	Specify custom code to include in the generated model terminate function.
"Include directories" on page 8-15	Specify a list of include folders to add to the include path.
"Source files" on page 8-17	Specify a list of additional source files to compile and link with the generated code.
"Libraries" on page 8-19	Specify a list of additional libraries to link with the generated code.
"Defines" on page 8-21	Specify preprocessor macro definitions to be added to the compiler command line.

See Also

More About

- “Model Configuration”

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 text to identify a folder, source file, or library.
3. Click **Apply**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Use the same custom code settings as Simulation Target

Description

Specify whether to use the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.

Category: Code Generation > Custom Code

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Source file

Description

Specify custom code to include near the top of the generated model source file.

Category: Code Generation > Custom Code

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: character vector
Value: C code
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2

- “Integrate External Code by Using Model Configuration Parameters”

Header file

Description

Specify custom code to include near the top of the generated model header file.

Category: Code Generation > Custom Code

Settings

Default:`''`

The code generator 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: character vector
Value: C code
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Initialize function

Description

Specify custom code to include in the generated model initialize function.

Category: Code Generation > Custom Code

Settings

Default: `''`

The code generator places code inside the model's initialize function in the *`model`*`.c` or *`model`*`.cpp` file.

Command-Line Information

Parameter: `CustomInitializer`
Type: character vector
Value: C code
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2

- “Integrate External Code by Using Model Configuration Parameters”

Terminate function

Specify custom code to include in the generated model terminate function.

Description

Specify custom code to include in the generated model terminate function.

Category: Code Generation > Custom Code

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.

Command-Line Information

Parameter: `CustomTerminator`
Type: character vector
Value: C code
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Include directories

Description

Specify a list of include folders to add to the include path.

Category: Code Generation > Custom Code

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 containing one or more spaces, you must enclose the path in double quotes. For example, the second and third paths 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 containing spaces must be separately double-quoted within the single-quoted third argument character vector, for example,

```
>> set_param('mymodel', 'CustomInclude', ...
             'C:\Project "C:\Custom Files" "C:\Library Files"')
```

Command-Line Information

Parameter: `CustomInclude`
Type: character vector

Value: folder path
Default: `' '`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Source files

Description

Specify a list of additional source files to compile and link with the generated code.

Category: Code Generation > Custom 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: character vector
Value: file name
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Libraries

Description

Specify a list of additional libraries to link with the generated code.

Category: Code Generation > Custom 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: character vector
Value: library file name
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Defines

Description

Specify preprocessor macro definitions to be added to the compiler command line.

Category: Code Generation > Custom Code

Settings

Default: `''`

Enter a list of macro definitions for the compiler command line. Specify the parameters with a space-separated list of macro definitions. If a makefile is generated, these macro definitions are added to the compiler command line in the makefile. The list can include simple definitions (for example, `-DDEF1`), definitions with a value (for example, `-DDEF2=1`), and definitions with a space in the value (for example, `-DDEF3="my value"`). Definitions can omit the `-D` (for example, `-DFOO=1` and `FOO=1` are equivalent). If the toolchain uses a different flag for definitions, the code generator overrides the `-D` and uses the appropriate flag for the toolchain.

Command-Line Information

Parameter: `CustomDefine`
Type: character vector
Value: preprocessor macro definition
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Integrate External Code by Using Model Configuration Parameters”

Code Generation Parameters: Interface

Model Configuration Parameters: Code Generation Interface

The **Code Generation > Interface** category includes parameters for configuring the interface of the generated code. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation > Interface** pane.

Parameter	Description
“Code replacement library” on page 9-11	Specify a code replacement library the code generator uses when producing code for a model.
“Shared code placement” on page 9-14	Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.
“Support: floating-point numbers” on page 9-16	Specify whether to generate floating-point data and operations.
“Support: non-finite numbers” on page 9-18	Specify whether to generate non-finite data and operations on non-finite data.
“Support: complex numbers” on page 9-20	Specify whether to generate complex data and operations.
“Support: absolute time” on page 9-22	Specify whether to generate and maintain integer counters for absolute and elapsed time values.
“Support: continuous time” on page 9-24	Specify whether to generate code for blocks that use continuous time.
“Support: variable-size signals” on page 9-27	Specify whether to generate code for models that use variable-size signals.
“Code interface packaging” on page 9-29	Select the packaging for the generated C or C++ code interface.

Parameter	Description
“Multi-instance code error diagnostic” on page 9-33	Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.
“Pass root-level I/O as” on page 9-35	Control how root-level model input and output are passed to the reusable *`model`*`_step` function.
“Remove error status field in real-time model data structure” on page 9-37	Specify whether to log or monitor error status.
“Array layout” on page 9-71	Specify layout of array data for code generation as column-major or row-major
“External functions compatibility for row-major code generation” on page 9-73	Select diagnostic action if Simulink encounters a function that has no specified array layout
“Parameter visibility” on page 9-39	Specify whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class.
“Parameter access” on page 9-41	Specify whether to generate access methods for block parameters for the C++ model class.
“External I/O access” on page 9-43	Specify whether to generate access methods for root-level I/O signals for the C ++ model class.
“Configure C++ Class Interface” on page 9-45	Customize the C++ class interface for your model code.
“Generate C API for: signals” on page 9-46	Generate C API data interface code with a signals structure.
“Generate C API for: parameters” on page 9-48	Generate C API data interface code with parameter tuning structures.
“Generate C API for: states” on page 9-50	Generate C API data interface code with a states structure.

Parameter	Description
“Generate C API for: root-level I/O” on page 9-52	Generate C API data interface code with a root-level I/O structure.
“ASAP2 interface” on page 9-54	Generate code for the ASAP2 data interface.
“External mode” on page 9-56	Generate code for the external mode data interface.
“Transport layer” on page 9-58	Specify the transport protocol for communications.
“MEX-file arguments” on page 9-61	Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.
“Static memory allocation” on page 9-64	Control memory buffer for external mode communication.
“Static memory buffer size” on page 9-66	Specify the memory buffer size for external mode communication.

These configuration parameters are under the **Advanced parameters**.

Parameter	Description
“Standard math library” on page 10-21	Specify the standard math library for your execution environment. 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 `C99 (ISO)` - ISO/IEC 9899:1999 C standard math library `C++03 (ISO)` - ISO/IEC 14882:2003 C++ standard math library
“Support non-inlined S-functions” on page 10-23	Specify whether to generate code for non-inlined S-functions.

Parameter	Description
“Maximum word length” on page 10-28	Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.
“Buffer size of dynamically-sized string (bytes)” on page 9-70	Number of bytes of the character buffer generated for dynamic string signals without maximum length.
“Multiword type definitions” on page 10-25	Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.
“Classic call interface” on page 10-30	Specify whether to generate model function calls compatible with the main program module of the GRT target in models created before R2012a.
“Use dynamic memory allocation for model initialization” on page 10-32	Control how the generated code allocates memory for model data.
“Single output/update function” on page 10-36	Specify whether to generate the *`model`*`_step` function.
“Terminate function required” on page 10-39	Specify whether to generate the *`model`*`_terminate` function.
“Combine signal/state structures” on page 10-41	Specify whether to combine global block signals and global state data into one data structure in the generated code
“Generate separate internal data per entry-point function” on page 10-87	Generate a model's block signals (block I/O) and discrete states (DWork) acting at the same rate into the same data structure.
“MAT-file logging” on page 10-50	Specify MAT-file logging.
“MAT-file variable name modifier” on page 10-53	Select the text to add to MAT-file variable names.
“Existing shared code” (Embedded Coder)	Specify folder that contains existing shared code

Parameter	Description
“Remove disable function” (Embedded Coder)	Remove unreachable (dead-code) instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies.
“Remove reset function” (Embedded Coder)	Remove unreachable (dead-code) instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies.
“LUT object struct order for even spacing specification” on page 9-68	Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to even spacing.
“LUT object struct order for explicit value specification” on page 9-69	Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to explicit value.
“Generate destructor” on page 10-48	Specify whether to generate a destructor for the C++ model class.
“Internal data access” on page 10-46	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.
“Internal data visibility” on page 10-44	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.
“Use dynamic memory allocation for model block instantiation” on page 10-34	Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ class interface.

Parameter	Description
“Code replacement library” on page 9-11	Create custom Code Replacement libraries using code replacement tool.
“Ignore custom storage classes” on page 10-2	Specify whether to apply or ignore custom storage classes.
“Ignore test point signals” on page 10-4	Specify allocation of memory buffers for test points.
“Implement each data store block as a unique access point” on page 10-85	Create unique variables for each read/write operation of a Data Store Memory block.
“Preserve Stateflow local data array dimensions” on page 9-75	Preserve the dimensions of multidimensional arrays in Stateflow local data in the generated code.

The following parameters under the **Advanced parameters** are infrequently used and have no other documentation.

Parameter	Description
`GenerateSharedConstants`	Control whether the code generator generates code with shared constants and shared functions. Default is `on`. `off` turns off shared constants, shared functions, and subsystem reuse across models.
`InferredTypesCompatibility`	For compatibility with legacy code including `tmwtypes.h`, specify that the code generator creates a preprocessor directive `#define_TMWTYPES_ inside rtwtypes.h`

Parameter	Description
`TargetLibSuffix` *`character vector`* - `''`	Control the suffix used for naming a target's dependent libraries (for example, `_target.lib` or `_target.a`). If specified, the character vector must include a period (.). (For generated model reference libraries, the library suffix defaults to `_rtwlib.lib` on Windows systems and `_rtwlib.a` on UNIX systems.). **Note** This parameter does not apply for model builds that use the toolchain approach, see “Library Control Parameters”
`TargetPreCompLibLocation` *`character vector`* - `''`	Control the location of precompiled libraries. If you do not set this parameter, the code generator uses the location specified in `rtwmakecfg.m`.
`IsERTTarget`	Indicates whether or not the currently selected target is derived from the ERT target.
`CPPClassGenCompliant`	Indicates whether the target supports the ability to generate and configure C++ class interfaces to model code.
`ConcurrentExecutionCompliant`	Indicates whether the target supports concurrent execution
`UseToolchainInfoCompliant`	Indicate a custom target is toolchain-compliant.
`ModelStepFunctionPrototypeControlCompliant`	Indicates whether the target supports the ability to control the function prototypes of initialize and step functions that are generated for a Simulink model.
`ParMdlRefBuildCompliant`	Indicates if the model is configured for parallel builds when building a model that includes referenced models.

Parameter	Description
`CompOptLevelCompliant` `off`, `on`	Set in `SelectCallback` for a target to indicate whether the target supports the ability to use the **Compiler optimization level** parameter to control the compiler optimization level for building generated code. Default is `off` for custom targets and `on` for targets provided with the Simulink Coder and Embedded Coder products.
`ModelReferenceCompliant` character vector - `off`, **`on`**	Set in `SelectCallback` for a target to indicate whether the target supports model reference.
`GenerateFullHeader`	Generate full header including time stamp. For ERT targets, this parameter is on the **Code Generation > Templates** pane.

The following parameters are for MathWorks use only.

Parameter	Description
`ExtModeTesting`	For MathWorks use only.
`ExtModeIntrfLevel`	For MathWorks use only.
`ExtModeMexFile`	For MathWorks use only.

See Also

More About

- “Model Configuration”

Code Generation: Interface Tab Overview

Select the target software environment, output variable name modifier, and data exchange interface.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Run-Time Environment Configuration”

Code replacement library

Description

Specify a code replacement library the code generator uses when producing code for a model.

Category: Code Generation > Interface

Settings

Default: `None`

`None`

Does not use a code replacement library.

`Named code replacement library`

Generates calls to a specific platform, compiler, or standards code replacement library. The list of named libraries depends on:

- Installed support packages.
- System target file, language, standard math library, and device vendor configuration.
- Whether you created and registered code replacement libraries, using the Embedded Coder product.

Tips

- Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.
- If you 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).

  - If you change the name or contents of the code replacement library and rebuild the model from the same folder as the previous build, the code generator reports a checksum warning (see “Manage the Shared Utility Code Checksum”). The warning prompts you to remove the existing folder and stop or stop code generation.
- If both of the following conditions exist for a model that contains Stateflow charts, the Simulink software regenerates code for the charts and recompiles the generated code.
  - You *do not* specify `Shared location` for the **Code Generation > Interface > Shared code placement** parameter.
  - You change the code replacement library name or contents before regenerating code.

Command-Line Information

Parameter: `CodeReplacementLibrary`
Type: character vector
Value: `'None'|'GNU C99 extensions'|'Intel IPP for x86-64 (Windows)'|'Intel IPP/SSE for x86-64 (Windows)'|'Intel IPP for x86-64 (Windows for MinGW compiler)'|'Intel IPP/SSE for x86-64 (Windows for MinGW compiler)'|'Intel IPP for x86/Pentium (Windows)'|'Intel IPP/SSE x86/Pentium (Windows)'|'Intel IPP for x86-64 (Linux)'|'Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Run-Time Environment Configuration”
- “What Is Code Replacement Customization?” (Embedded Coder)
- “Develop a Code Replacement Library” (Embedded Coder)

Shared code placement

Description

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Category: Code Generation > Interface

Settings

Default: `Auto`

`Auto`

The code generator places utility code within the *`codeGenFolder`*`/slprj/`*`target`*`/_sharedutils` (or *`codeGenFolder`*`/`*`targetSpecific`*`/_shared`) folder for a model that contains Existing Shared Code (Embedded Coder) or at least one of the following blocks:

- Model blocks
- Simulink Function blocks
- Function Caller blocks
- Calls to Simulink Functions from Stateflow or MATLAB Function blocks
- Stateflow graphical functions when the **Export Chart Level Functions** parameter is selected

If a model does not contain one of the above blocks or Existing Shared Code (Embedded Coder), the code generator places utility code in the build folder (generally, the folder that contains *`model`*`.c` or *`model`*`.cpp`).

`Shared location`

Directs code for utilities to be placed within the *`codeGenFolder`*`/slprj/`*`target`*`/_sharedutils` (or *`codeGenFolder`*`/`*`targetSpecific`*`/_shared`) folder.

Command-Line Information

Parameter: `UtilityFuncGeneration`
Type: character vector

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Run-Time Environment Configuration”
- “Manage Build Process Folders”
- “Sharing Utility Code”

Support: floating-point numbers

Description

Specify whether to generate floating-point data and operations.

Category: Code Generation > Interface

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.
- When you select the configuration parameter **MAT-File Logging**, you must also select **Support: non-finite numbers** and **Support: floating-point numbers**.

Command-Line Information

Parameter: `PurelyIntegerCode`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Note The command-line values are reverse of the settings values. The value `'on'` in the command line corresponds to the description of “Off” in the settings section. The value `'off'` in the command line corresponds to the description of “On” in the settings section.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (GUI), `'on'` (command-line) — for integer only
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

Support: non-finite numbers

Description

Specify whether to generate non-finite data and operations on non-finite data.

Category: Code Generation > Interface

Settings

Default: on

 On

Generates non-finite data (for example, `NaN` and `Inf`) and related operations.

 Off

Does not generate non-finite data and operations. If you clear this option, an error occurs if the code generator encounters non-finite data or expressions. The error message reports offending blocks and parameters.

Note Code generation is optimized with the assumption that non-finite data are absent. However, if your application produces non-finite numbers through signal data or MATLAB code, the behavior of the generated code might be inconsistent with simulation results when processing non-finite data.

Dependencies

- For ERT-based targets, parameter **Support: floating-point numbers** enables **Support: non-finite numbers**.
- If `off` for top model, must be `off` for referenced models.
- When you select the configuration parameter **MAT-File Logging**, you must also select **Support: non-finite numbers** and, if you use an ERT-based system target file, **Support: floating-point numbers**.

Command-Line Information

Parameter: `SupportNonFinite`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

Support: complex numbers

Description

Specify whether to generate complex data and operations.

Category: Code Generation > Interface

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.
- If `off` for top model, must be `off` for referenced models.

Command-Line Information

Parameter: `SupportComplex`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (for real only)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

Support: absolute time

Description

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

Category: Code Generation > Interface

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.
- Select this parameter if your model includes blocks that require absolute or elapsed time values.

Command-Line Information

Parameter: `SupportAbsoluteTime`
Type: character vector

Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Timers in Asynchronous Tasks”

Support: continuous time

Description

Specify whether to generate code for blocks that use continuous time.

Category: Code Generation > Interface

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 parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license to generate code.
- This parameter must be on for models that include blocks that require absolute or elapsed time values.
- This parameter is cleared if you select **Remove error status field in real-time model data structure**.
- If both the following conditions exist, output values read from `ert_main` for a continuous output port can differ from the corresponding output values in logged data for a model:
  - You customize `ert_main.c` or `.cpp` to read model outputs after each base-rate model step.
  - You select parameters **Support: continuous time** and **Single output/update function**.

The difference occurs because, while logged data captures output at major time steps, output read from `ert_main` after the base-rate model step can capture output at intervening minor time steps. The following table lists workarounds that eliminate the discrepancy.

Work Around	Customized ert_main.c	Customized ert_main.cpp
Separate the generated output and update functions (clear the **Single output/update function** parameter), 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`).	X	
Select the **Single output/update function** parameter. Insert code in the generated *`model`*`.c` or `.cpp` file that returns model output values reflecting only major time steps. For example, in the model step function, between the output code and the update code, save the value of the model `External outputs` global data structure (defined in *`model`*`.h`). Then, restore the value after the update code completes.	X	X
Place a Zero-Order Hold block before the continuous output port.	X	X

Command-Line Information

Parameter: `SupportContinuousTime`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Use Discrete and Continuous Time” (Embedded Coder)

Support: variable-size signals

Description

Specify whether to generate code for models that use variable-size signals.

Category: Code Generation > Interface

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

Code interface packaging

Description

Select the packaging for the generated C or C++ code interface.

Category: Code Generation > 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 **Use dynamic memory allocation for model initialization** 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 handwritten code, add an `#include model.h` directive to the code.
- When you select `Reusable function`, the code generator generates a pointer to the real-time model object (*`model`*`_M`).
- When you select `Reusable function`, the code generator can 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 **Use dynamic memory allocation for model initialization**.
- To enable **Classic call interface**, you must select `Nonreusable function`.
- 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 dynamic memory allocation for model block instantiation**
- For an ERT target, you can use `Reusable function` with the static `ert_main.c` module, if you do the following:
  - Select the value `Part of model data structure` for **Pass root-level I/O as**.
  - Select the option **Use dynamic memory allocation for model initialization**.
- 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 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 change the code generated for function-call subsystems.

Command-Line Information

Parameter: `CodeInterfacePackaging`
Type: character vector
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

`model_step`

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Generation for Model Entry-Point Functions”
- “Generate Reentrant Code from Top Models”

- “Combine Code Generated for Multiple Models or Multiple Instances of a Model”
- “Generate Reentrant Code from Top-Level Models” (Embedded Coder)
- “Generate C++ Class Interface to Model or Subsystem Code”
- “Customize Generated C++ Class Interfaces” (Embedded Coder)
- “Control Generation of Subsystem Functions”
- “Generate Reentrant Code from Subsystems”
- “S-Functions That Support Code Reuse”
- “Static Main Program Module” (Embedded Coder)
- “Customize Generated C Function Interfaces” (Embedded Coder)
- “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder)
- “Generate Component Source Code for Export to External Code Base” (Embedded Coder)

Multi-instance code error diagnostic

Description

Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.

Category: Code Generation > Interface

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 can:

- Generate code that compiles but is not reentrant. For example, if a signal or DWork structure 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 code generator produces 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: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Generation for Model Entry-Point Functions”
- “Generate Reentrant Code from Top Models”
- “Generate C++ Class Interface to Model or Subsystem Code”
- “Control Generation of Subsystem Functions”
- “Generate Reentrant Code from Subsystems”
- “Generate Reentrant Code from Subsystems”
- “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder)

Pass root-level I/O as

Description

Control how root-level model input and output are passed to the reusable *`model`*`_step` function.

Category: Code Generation > Interface

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 `Reusable function`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `RootIOFormat`
Type: character vector
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

`model_step`

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Generation for Model Entry-Point Functions”
- “Generate Reentrant Code from Top-Level Models” (Embedded Coder)
- “Control Generation of Subsystem Functions”
- “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder)

Remove error status field in real-time model data structure

Description

Specify whether to log or monitor error status.

Category: Code Generation > Interface

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.
- 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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Use the Real-Time Model Data Structure”

Parameter visibility

Description

Specify whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class.

Category: Code Generation > Interface

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: character vector
Value: `'public'` | `'private'` | `'protected'`
Default: `'private'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options” (Embedded Coder)

Parameter access

Description

Specify whether to generate access methods for block parameters for the C++ model class.

Category: Code Generation > Interface

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: character vector
Value: `'None'` | `'Method'` | `'Inlined method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`
Efficiency	`Inlined method`
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options” (Embedded Coder)

External I/O access

Description

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. The parameter has *no* effect 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 Default Arguments” (Embedded Coder) and “Passing I/O Arguments” (Embedded Coder).

Category: Code Generation > Interface

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: character vector
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	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options” (Embedded Coder)

Configure C++ Class Interface

Description

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.

Category: Code Generation > 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” (Embedded Coder).

See Also

`model_step`

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Customize Generated C++ Class Interfaces” (Embedded Coder)
- “Configure Step Method for Your Model Class” (Embedded Coder)

Generate C API for: signals

Description

Generate C API data interface code with a signals structure.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates C API interface to global block outputs.

 Off

Does not generate C API signals.

Command-Line Information

Parameter: `RTWCAPISignals`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Exchange Data Between Generated and External Code Using C API”

Generate C API for: parameters

Description

Generate C API data interface code with parameter tuning structures.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates C API interface to global block parameters.

 Off

Does not generate C API parameters.

Command-Line Information

Parameter: `RTWCAPIParams`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Exchange Data Between Generated and External Code Using C API”

Generate C API for: states

Description

Generate C API data interface code with a states structure.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates C API interface to discrete and continuous states.

 Off

Does not generate C API states.

Command-Line Information

Parameter: `RTWCAPIStates`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Exchange Data Between Generated and External Code Using C API”

Generate C API for: root-level I/O

Description

Generate C API data interface code with a root-level I/O structure.

Category: Code Generation > Interface

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.

Command-Line Information

Parameter: `RTWCAPIRootIO`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Exchange Data Between Generated and External Code Using C API”

ASAP2 interface

Description

Generate code for the ASAP2 data interface.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates code for the ASAP2 data interface.

 Off

Does not generate code for the ASAP2 data interface.

Command-Line Information

Parameter: `GenerateASAP2`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Export ASAP2 File for Data Measurement and Calibration”

External mode

Description

Generate code for the external mode data interface.

Category: Code Generation > Interface

Settings

Default: off

 On

Generates code for the external mode data interface.

 Off

Does not generate code for the external mode data interface.

Command-Line Information

Parameter: `ExtMode`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development Off for production code generation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Host-Target Communication with External Mode Simulation”
- “External Mode Simulation with XCP Communication”
- “Host-Target Communication with External Mode Simulation”

Transport layer

Description

Specify the transport protocol for communications.

Category: Code Generation > Interface

Settings

Default: `tcpip`

`tcpip`

Use a TCP/IP transport mechanism. Selecting this option sets **MEX-file name** to `ext_comm`.

`serial`

Use a serial transport mechanism. Selecting this option sets **MEX-file name** to `ext_serial_win32_comm`.

`XCP on TCP/IP`

Use XCP protocol with TCP/IP transport layer. Selecting this option sets **MEX-file name** to `ext_xcp`.

`XCP on Serial`

Use XCP protocol with serial transport layer. Selecting this option sets **MEX-file name** to `ext_xcp`.

`customTransportLayer`

Use custom transport layer.

Tips

The Configuration Parameters dialog box displays **MEX-file name** next to **Transport layer**. You cannot edit the value in the **MEX-file name** field. 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 transport mechanisms.

The command-line parameter is an index. To get the transport layer index, use these commands:

```
cs = getActiveConfigSet(modelName);
index = Simulink.ExtMode.Transports.getExtModeTransportIndex(cs, transportLayer);
```

`transportLayer` is one of these values:

- `'tcpip'`
- `'serial'`
- `'XCP on TCP/IP'`
- `'XCP on Serial'`
- *`customTransportLayer`*

To select the transport layer:

```
set_param(cs, 'ExtModeTransport', index)
```

To determine the transport layer:

```
transportLayerName = Simulink.ExtMode.Transports.getExtModeTransport(cs, index)
```

Dependency

Selecting **External mode** enables this parameter.

Command-Line Information

Parameter: `ExtModeTransport`
Type: integer
Value: See “Tips” on page 9-58
Default: `0`

Recommended Settings

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

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “External Mode Simulation with XCP Communication”
- “External Mode Simulation with TCP/IP or Serial Communication”
- “Customize XCP Slave Software”
- “Create a Transport Layer for TCP/IP or Serial External Mode Communication”

MEX-file arguments

Description

Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.

Category: Code Generation > Interface

Settings

Default: `''`

For XCP communication with a TCP/IP transport layer, there are four optional arguments:

- Network name of your target processor –– For example, `'localhost'` if the target process is your development computer or the IP address `'148.27.151.12'`.
- Verbosity level –– `0` for no information or `1` to display detailed information during data transfer.
- Port number of TCP/IP server –– An integer value between `256` and `65535`, with a default of `17725`.
- Symbols file name –– File format is PDB for Windows or ELF for Linux®.

For XCP communication with a serial transport layer, there are four optional arguments:

- Verbosity level –– `0` for no information or `1` for detailed information.
- Serial port ID –– On Windows, `'COM1'` or 1 for `COM1`, `'COM2'` or 2 for `COM2`, etc. On Linux, `'/dev/ttyS0'`, etc.
- Baud –– `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600` (default), or `115200`.
- Symbols file name –– File format is PDB for Windows or ELF for Linux.

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target processor –– For example, `'myComputer'` or `'148.27.151.12'`.
- Verbosity level –– `0` for no information or `1` to display detailed information during data transfer.

- Port number of TCP/IP server –– 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 –– `1` for `COM1`, etc.
- Baud –– `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600` (default), or `115200`.

Specify the arguments in the list order. For example, if you want to specify the verbosity level (the second argument), then you must also specify the network name of the target processor (the first argument). You can use white space or commas as argument delimiters.

```
'148.27.151.12' 1 30000
```

Dependency

Selecting **External mode** enables this parameter.

Command-Line Information

Parameter: `ExtModeMexArgs`
Type: character vector
Value: valid arguments
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “External Mode Simulation with XCP Communication”
- “External Mode Simulation with TCP/IP or Serial Communication”

Static memory allocation

Description

Control memory buffer for external mode communication.

Category: Code Generation > Interface

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 to allocate, select verbose mode on the target. That selection displays the amount of memory the target tries to allocate and the amount of memory available.

Dependencies

- Selecting **External mode** enables this parameter.
- This parameter enables **Static memory buffer size**.
- This parameter is ignored if **Transport layer** is `XCP on TCP/IP` or `XCP on Serial`. The platform abstraction layer of the XCP slave software provides a configurable memory allocator.

Command-Line Information

Parameter: `ExtModeStaticAlloc`
Type: character vector

Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “External Mode Simulation with TCP/IP or Serial Communication”
- “Control Memory Allocation for Communication Buffers in Target”
- “XCP Platform Abstraction Layer”

Static memory buffer size

Description

Specify the memory buffer size for external mode communication.

Category: Code Generation > Interface

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 to allocate, select verbose mode on the target. That selection displays the amount of memory the target tries to allocate and the amount of memory available.

Dependency

Selecting **Static memory allocation** enables this parameter.

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “External Mode Simulation with TCP/IP or Serial Communication”
- “Control Memory Allocation for Communication Buffers in Target”

LUT object struct order for even spacing specification

Description

Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to even spacing.

Category: Code Generation > Interface > Advanced Parameters

Settings

Default: `Size,Breakpoints,Table`

`Size,Breakpoints,Table`

Display structure in the order size, breakpoints, table.

`Size,Table,Breakpoints`

Display structure in the order size, table, breakpoints.

Command-Line Information

Parameter: `LUTObjectStructOrderEvenSpacing`
Type: character vector
Value: `'Size,Breakpoints,Table'` | `'Size,Table,Breakpoints'`
Default: `'Size,Breakpoints,Table'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

LUT object struct order for explicit value specification

Description

Change the order of the fields in the generated structure for a lookup table object whose specification parameter is set to explicit value.

Category: Code Generation > Interface> Advanced Parameters

Settings

Default: `Size,Breakpoints,Table`

`Size,Breakpoints,Table`

Display structure in the order size, breakpoints, table.

`Size,Table,Breakpoints`

Display structure in the order size, table, breakpoints.

Command-Line Information

Parameter: `LUTObjectStructOrderExplicitValues`
Type: character vector
Value: `'Size,Breakpoints,Table'` | `'Size,Table,Breakpoints'`
Default: `'Size,Breakpoints,Table'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Buffer size of dynamically-sized string (bytes)

Description

Number of bytes of the character buffer generated for dynamic string signals without maximum length.

Category: Code Generation > Interface

Settings

Default: 256

Command-Line Information

Parameter: `DynamicStringBufferSize`
Type: character vector
Value: scalar
Default: 256

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Array layout

Description

Specify the layout of array data for code generation as column-major or row-major.

Category: Code Generation > Interface

Settings

Default: `Column-major`

`Column-major`

Generate code in column-major array layout. For example, consider matrix `A`, which is a 4x3 matrix:

```
A =
    1    2    3
    4    5    6
    7    8    9
    10   11   12
```

In column-major array layout, the elements of the columns are contiguous in memory. `A` is represented in the generated code as:

```
1    4    7    10    2    5    8    11    3    6    9    12
```

`Row-major`

Generate code in row-major array layout. For example, for matrix `A`, in row-major array layout, the elements of the rows are contiguous. `A` is represented in the generated code as:

```
1    2    3    4    5    6    7    8    9    10    11    12
```

Select the configuration parameter **Use algorithms optimized for row-major array layout** (Simulink) to enable efficient row-major algorithms.

Command-Line Information

Parameter: `ArrayLayout`
Type: character vector

Value: `'Column-major'` | `'Row-major'`
Default: `'Column-major'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Code Generation of Matrices and Arrays”

External functions compatibility for row-major code generation

Description

Select the diagnostic action if Simulink encounters a function that has no specified array layout.

Category: Code Generation > Interface

Settings

Default: `error`

`none`

The code generator generates the code without any error or warning.

`warning`

The code generator builds the model and displays a warning message.

`error`

The code generator terminates the build and displays an error message.

Dependencies

To enable this parameter, set **Array layout** on page 9-71 to `Row-major`.

This parameter works only if your S-function or custom C function has a multidimensional array.

Command-Line Information

Parameter: `UnsupportedSFcnMsg`
Type: character vector
Value: `'none'` | `'warning'` | `'error'`
Default: `'error'`

Recommended Settings

Application	Setting
Debugging	`warning`
Traceability	No impact
Efficiency	No impact
Safety precaution	`error`

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Generate Row-Major Code for S-Functions”
- “Code Generation of Matrices and Arrays”

Preserve Stateflow local data array dimensions

Description

Preserve the dimensions of multidimensional arrays in Stateflow local data in the generated code.

Category: Code Generation > Interface

Settings

Default: Off

 On

Preserves dimensions of multidimensional arrays in the generated code.

 Off

Flattens the multidimensional array to one dimension in the generated code.

Dependencies

- To enable this parameter, set **Array layout** on page 9-71 to `Row-major`.
- This parameter works only if your model has a Stateflow chart.
- This parameter requires Embedded Coder.

Command-Line Information

Parameter: `PreserveStateflowLocalDataDimensions`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Preserve Dimensions of Multidimensional Arrays in Generated Code” (Embedded Coder)
- “Select Array Layout for Matrices in Generated Code” (Stateflow)

10

Simulink Coder Parameters: Advanced Parameters

Ignore custom storage classes

Description

Specify whether to apply or ignore custom storage classes.

Category: Code Generation > Interface

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: character vector
Value: `'on'` | `'off`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements” (Embedded Coder)

Ignore test point signals

Description

Specify allocation of memory buffers for test points.

Category: Code Generation > Interface

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Appearance of Test Points in the Generated Code”
- “Test Points” (Simulink)
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Code-to-model

Description

Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.

Category: Code Generation > Report

Settings

Default: On

 On

Includes hyperlinks in the code generation report that link code to corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram. The hyperlinks provide traceability for validating generated code against the source model.

 Off

Omits hyperlinks from the generated report.

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** on page 5-5.
- You must select **Include comments** on page 6-5 on the **Code Generation** > **Comments** pane to use this parameter.

Command-Line Information

Parameter: `IncludeHyperlinkInReport`
Type: character vector
Value: `'on'` | `'off`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions” (Embedded Coder)

Model-to-code

Description

Link Simulink blocks, Stateflow objects, and MATLAB functions in a model diagram to corresponding code segments in a generated HTML report so that the generated code for a block can be highlighted on request.

Category: Code Generation > Report

Settings

Default: On

 On

Includes model-to-code highlighting support in the code generation report. To highlight the generated code for a Simulink block, Stateflow object, or MATLAB script in the code generation report, right-click the item and select **C/C++ Code** > **Navigate to C/C++ Code**.

 Off

Omits model-to-code highlighting support from the generated 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** on page 5-5.
- You must select the following parameters to use this parameter:
  - **Include comments** on page 6-5 on the **Code Generation > Comments** pane
  - At least one of the following:
    - **Eliminated / virtual blocks** on page 10-11
    - **Traceable Simulink blocks** on page 10-13

- **Traceable Stateflow objects** on page 10-15
- **Traceable MATLAB functions** on page 10-17

Command-Line Information

Parameter: `GenerateTraceInfo`
Type: Boolean
Value: `on` | `off`
Default: `on`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions” (Embedded Coder)

Configure

Description

Open the Model-to-code navigation dialog box. Through this dialog box you specify a build folder containing previously generated model code to highlight. When you apply your build folder selection, Embedded Coder attempts to load traceability information from the earlier build, if you selected **Model-to-code** on page 10-8 parameter.

Category: Code Generation > Report

Dependency

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Model-to-code** parameter.

See Also

Related Examples

- “Reload Existing Traceability Information” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Report” on page 5-2

Eliminated / virtual blocks

Description

Include summary of eliminated and virtual blocks in the **Traceability Report** section of the code generation report.

Category: Code Generation > Report

Settings

Default: On

 On

Include summary of eliminated and virtual blocks in the **Traceability Report** section of the code generation report.

 Off

Does not include a summary of eliminated and virtual blocks.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** on page 5-5 parameter.

Command-Line Information

Parameter: `GenerateTraceReport`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions” (Embedded Coder)
- “Customize Traceability Reports” (Embedded Coder)

Traceable Simulink blocks

Description

Category: Code Generation > Report

Includes a summary of Simulink blocks and the corresponding code locations in the **Traceability Report** section of the code generation report.

Settings

Default: On

 On

Includes a summary of Simulink blocks and the corresponding code locations in the **Traceability Report** section of the code generation report.

 Off

Does not include a summary of Simulink blocks.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** on page 5-5 parameter.

Command-Line Information

Parameter: `GenerateTraceReportSl`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions” (Embedded Coder)
- “Customize Traceability Reports” (Embedded Coder)

Traceable Stateflow objects

Description

Includes a summary of Stateflow objects and the corresponding code locations in the **Traceability Report** section of the code generation report.

Category: Code Generation > Report

Settings

Default: On

 On

Includes a summary of Stateflow objects and the corresponding code locations in the **Traceability Report** section of the code generation report.

 Off

Does not include a summary of Stateflow objects.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** on page 5-5 parameter.

Command-Line Information

Parameter: `GenerateTraceReportSf`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions” (Embedded Coder)
- “Customize Traceability Reports” (Embedded Coder)
- “Traceability of Stateflow Objects in Generated Code” (Stateflow)

Traceable MATLAB functions

Description

Includes a summary of MATLAB functions and corresponding code locations in the **Traceability Report** section of the code generation report.

Category: Code Generation > Report

Settings

Default: On

 On

Includes a summary of MATLAB functions and corresponding code locations in the **Traceability Report** section of the code generation report.

 Off

Does not include a summary of MATLAB functions.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.
- Is enabled by the **Create code generation report** on page 5-5 parameter.

Command-Line Information

Parameter: `GenerateTraceReportEml`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions” (Embedded Coder)
- “Customize Traceability Reports” (Embedded Coder)

Summarize which blocks triggered code replacements

Description

Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

Category: 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** on page 5-5.

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

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- Analyze Code Replacements in the Generated Code (Embedded Coder)
- Trace Code Replacements Generated Using Your Code Replacement Library (Embedded Coder)
- Determine Why Code Replacement Functions Were Not Used (Embedded Coder)

Standard math library

Description

Specify standard math library for model.

Category: Code Generation > Interface

Settings

Default: `C99 (ISO)` or, if **Language** is set to `C++`, `C++03 (ISO)`

`C89/C90 (ANSI)`

Generates calls to the ISO/IEC 9899:1990 C standard math library.

`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.

Tips

- 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.
- If you are using a compiler that does not support ISO/IEC 9899:1999 C, set this parameter to `C89/C90 (ANSI)`.
- The build process checks whether the specified standard math library and toolchain are compatible. If they are not compatible, a warning occurs during code generation and the build process continues.

Dependencies

- `C++03` is available for use only if you select `C++` for the **Language** parameter.
- When you change the value of the **Language** parameter, the standard math library updates to `C99 (ISO)` for C and `C++03 (ISO)` for C++.

Command-Line Information

Parameter: `TargetLangStandard`
Type: character vector
Value: `'C89/C90 (ANSI)'` | `'C99 (ISO)'` | `'C++03 (ISO)'`
Default: For C, `'C99 (ISO)'`; for C++ `'C++03 (ISO)'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Specify Single-Precision Data Type for Embedded Application”
- “Run-Time Environment Configuration”

Support non-inlined S-functions

Description

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

Category: Code Generation > Interface

Settings

Default: Off

 On

Generates code for non-inlined S-functions.

 Off

Does not generate code for non-inlined S-functions. If this parameter is off and the model includes a non-inlined 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.
- Non-inlined S-functions require additional memory and computation resources, and can result in significant performance issues. Consider using an inlined S-function when efficiency is a concern.

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “S-Functions and Code Generation”

Multiword type definitions

Description

Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.

Category: Code Generation > Interface

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: character vector
Value: `'System defined'` | `'User defined'`
Default: `'System defined'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

Maximum word length

Description

Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.

Category: Code Generation > Interface

Settings

Default: `256` for ERT targets, `2048` for GRT targets

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 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` for ERT targets, `2048` for GRT targets

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

Classic call interface

Description

Specify whether to generate model function calls compatible with the main program module of the GRT target in models created before R2012a.

Category: Code Generation > Interface

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 system target file (`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
- **Code interface packaging** set to `Reusable function` or `C++ class`.
- Nonvirtual subsystem option **Function with separate data**

Dependencies

- Setting **Code interface packaging** to `C++ class` disables this option.

- 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: character vector
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	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Use Discrete and Continuous Time” (Embedded Coder)

Use dynamic memory allocation for model initialization

Description

Control how the generated code allocates memory for model data.

Category: Code Generation > Interface

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

`model_step`

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Generation for Model Entry-Point Functions”
- “Generate Reentrant Code from Top-Level Models” (Embedded Coder)
- “Control Generation of Subsystem Functions”
- “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder)

Use dynamic memory allocation for model block instantiation

Description

Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ class interface.

Category: Code Generation > 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 dynamic memory allocation for model block instantiation** 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” (Embedded Coder).

☐ 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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options” (Embedded Coder)

Single output/update function

Description

Specify whether to generate the *model*_step function.

Category: Code Generation > Interface

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. For more information about direct feed through, see “Algebraic Loops” (Simulink).

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

- Setting **Code interface packaging** to `C++ class` forces on and disables this option.
- This option and **Classic call interface** are mutually incompatible and cannot both be selected through the GUI. Selecting **Classic call interface** forces off and disables this option and clearing **Classic call interface** enables (but does not select) this option.
- When you use this option, you must clear the option **Minimize algebraic loop occurrences** on the **Model Referencing** pane.
- If you customize `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, selecting both parameters **Support: continuous time** and **Single output/update function** can 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. The following table lists workarounds that eliminate the discrepancy.

Work Around	Customized ert_main.c	Customized ert_main.cpp
Separate the generated output and update functions (clear the **Single output/update function** parameter), 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`).	X	
Select the **Single output/update function** parameter and insert code in the generated *`model`*`.c` or `.cpp` file to return model output values reflecting only major time steps. For example, in the model step function, between the output code and the update code, save the value of the model `External outputs` global data structure (defined in *`model`*`.h`). Then, restore the value after the update code completes.	X	X

Work Around	Customized ert_main.c	Customized ert_main.cpp
Place a Zero-Order Hold block before the continuous output port.	X	X

Command-Line Information

Parameter: `CombineOutputUpdateFcns`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “rt_OneStep and Scheduling Considerations” (Embedded Coder)

Terminate function required

Description

Specify whether to generate the *model*_terminate function.

Category: Code Generation > Interface

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: character vector
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

`model_terminate`

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

Combine signal/state structures

Description

Specify whether to combine global block signals and global state data into one data structure in the generated code

Category: Code Generation > Interface

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: character vector
Value: `'on'` | `'off'`
Default: `off`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “How Generated Code Stores Internal Signal, State, and Parameter Data”

Internal data visibility

Description

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.

Category: Code Generation > Interface

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: character vector
Value: `'public'` | `'private'` | `'protected'`
Default: `'private'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options” (Embedded Coder)

Internal data access

Description

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.

Category: Code Generation > Interface

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: character vector
Value: `'None'` | `'Method'` | `'Inlined method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`
Efficiency	`Inlined method`
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options” (Embedded Coder)

Generate destructor

Description

Specify whether to generate a destructor for the C++ model class.

Category: Code Generation > Interface

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options” (Embedded Coder)

MAT-file logging

Description

Specify MAT-file logging

Category: Code Generation > Interface

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** (see “Model Configuration Parameters: Data Import/Export” (Simulink))
- To Workspace blocks
- To File blocks
- Scope blocks with the **Log data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in “Export Simulation Data” (Simulink) 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

- When you select **MAT-file logging**, you must also select the configuration parameters **Support: non-finite numbers** and, if you use an ERT-based system target file, **Support: floating-point numbers**.
- Selecting this option enables **MAT-file variable name modifier**.
- For ERT-based system target files, clear this parameter if you are using exported function calls.

Limitations

- The code generator does not support MAT-file logging for custom data types (data types that are not built into Simulink).
- MAT-file logging does not support file-scoped data, for example, data items to which you apply the built-in custom storage class `FileScope`.
- 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++®, Texas Instruments™ Code Composer Studio™, Wind River® DIAB/GCC.
- MAT-file logging does not support Outport blocks to which you apply the storage class `ImportedExternPointer` or custom storage classes that yield nonaddressable data in the generated code. For example, the custom storage class `GetSet` causes the Outport to appear in the generated code as a function call, which is not addressable. This limitation applies whether you apply the storage class directly by using, for example, the Model Data Editor, or by resolving the Outport to a `Simulink.Signal` object that uses the storage class. As a workaround, apply the storage class to the signal that enters the Outport block.

Command-Line Information

Parameter: `MatFileLogging`
Type: character vector
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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Log Program Execution Results”
- “Log Data for Analysis”
- “Virtualized Output Ports Optimization” (Embedded Coder)
- “Virtualized Output Ports Optimization” (Embedded Coder)

MAT-file variable name modifier

Description

Select the text to add to MAT-file variable names.

Category: Code Generation > Interface

Settings

Default: `rt_`

`rt_`
: Adds prefix text.

`_rt`
: Adds suffix text.

`none`
: Does not add text.

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: character vector
Value: `'none'|'rt_'|'_rt'`
Default: `'rt_'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Log Program Execution Results”
- “Log Data for Analysis”

Verbose build

Description

Display code generation progress.

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Debug”

Retain .rtw file

Description

Specify *`model`*`.rtw` file retention.

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Debug”

Profile TLC

Description

Profile the execution time of TLC files.

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Debug”

Start TLC debugger when generating code

Description

Specify use of the TLC debugger

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Debug”

Start TLC coverage when generating code

Description

Generate the TLC execution report.

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Debug”

Enable TLC assertion

Description

Produce the TLC stack trace

Category: Code Generation

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Debug”

Custom FFT library callback

Description

Specify a callback class for FFTW library calls in code generated for FFT functions in MATLAB code. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object™ associated with a MATLAB System block.

To improve the execution speed of FFT functions, the code generator produces calls to the FFTW library that you specify in the callback class.

Category: Code Generation

Settings

Default: `''`

Specify the name of an FFT library callback class. If this parameter is empty, the code generator uses its own algorithms for FFT functions instead of calling the FFTW library.

Limitation

The class definition file must be in a folder on the MATLAB path.

Tip

Specify only the name of the class. Do not specify the name of the class definition file.

Command-Line Information

Parameter: `CustomFFTCallback`
Type: character vector
Value: class name
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

More About

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block”

External Websites

- www.fftw.org

Custom BLAS library callback

Description

Specify BLAS library callback class for BLAS calls in code generated from MATLAB code. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block.

Category: Code Generation

Settings

Default: `''`

Specify the name of a BLAS callback class that derives from `coder.BLASCallback`. If you specify a BLAS callback class, for certain low-level vector and matrix operations, the code generator produces BLAS calls by using the CBLAS C interface to your BLAS library. The callback class provides the CBLAS header and data type information and the information required to link to your BLAS library. If this parameter is empty, the code generator produces code for the vector and matrix functions instead of a BLAS call.

Limitation

The class definition file must be in a folder on the MATLAB path.

Tip

Specify only the name of the class. Do not specify the name of the class definition file.

Command-Line Information

Parameter: `CustomBLASCallback`
Type: character vector
Value: class name
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Speed Up Matrix Operations in Code Generated from a MATLAB Function Block”

Custom LAPACK library callback

Description

Specify LAPACK library callback class for LAPACK calls in code generated from MATLAB code. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block.

Category: Code Generation

Settings

Default: `''`

Specify the name of a LAPACK callback class that derives from `coder.LAPACKCallback`. If you specify a LAPACK callback class, for certain linear algebra functions, the code generator produces LAPACK calls by using the LAPACKE C interface to your LAPACK library. The callback class provides the name of your LAPACKE header file and the information required to link to your LAPACK library. If this parameter is empty, the code generator produces code for linear algebra functions instead of a LAPACK call.

Limitation

The class definition file must be in a folder on the MATLAB path.

Tip

Specify only the name of the class. Do not specify the name of the class definition file.

Command-Line Information

Parameter: `CustomLAPACKCallback`
Type: character vector
Value: class name
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Speed Up Linear Algebra in Code Generated from a MATLAB Function Block”

Shared checksum length

Description

Specify character length of `$C` token.

Category: Code Generation > Symbols

Settings

Default: 8 **Minimum:** 1 **Maximum:** 15

Specify an integer value that indicates the number of characters to expand the `$C` token for the **Shared utilities identifier format** parameter.

Tip

To avoid the possibility of a naming collision, consider increasing this parameter value.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `SharedChecksumLength`
Type: integer
Value: valid value
Default: 8

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Shared utilities identifier format” on page 7-28

EMX array utility functions identifier format

Description

Customize generated identifiers for `emxArray` (embeddable `mxArray`) utility functions. The code generator produces `emxArray` types for variable-size arrays that use dynamically allocated memory. It produces `emxArray` utility functions that create and interact with variables that have an `emxArray` type. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block. This parameter does not apply to:

- Input or output signals
- Parameters
- Global variables
- Discrete state properties of System objects associated with a MATLAB System block

Category: Code Generation > Symbols

Settings

Default: `emx$M$N`

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

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert the utility function name into identifier. For example, `Init_real`.
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- The code generator applies the identifier format specified by this parameter before it applies the formats specified by other identifier format control parameters.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- If you specify `$R`, the value that you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires an Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrEmxFcn`
Type: character vector
Value: valid combination of tokens
Default: `emx$M$N`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

EMX array types identifier format

Description

Customize generated identifiers for `emxArray` (embeddable `mxArray`) types. The code generator produces `emxArray` types for variable-size arrays that use dynamically allocated memory. This parameter applies to MATLAB code in a MATLAB Function block, a Stateflow chart, or a System object associated with a MATLAB System block. This parameter does not apply to:

- Input or output signals
- Parameters
- Global variables
- Discrete state properties of System objects associated with a MATLAB System block

Category: Code Generation > Symbols

Settings

Default: `emxArray_$M$N`

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

Token	Description
`$M`	Insert name-mangling text if required to avoid naming collisions. Required.
`$N`	Insert type name. For example, `real_T`
`$R`	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- The code generator applies the identifier format specified by this parameter before it applies the formats specified by other identifier format control parameters.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for name-mangling text.
- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires an Embedded Coder when generating code.

Command-Line Information

Parameter: `CustomSymbolStrEmxType`
Type: character vector
Value: valid combination of tokens
Default: `emxArray_$M$N`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control” (Embedded Coder)
- “Control Name Mangling in Generated Identifiers” (Embedded Coder)
- “Identifier Format Control Parameters Limitations” (Embedded Coder)

Use Simulink Coder Features

Description

Enable “Simulink Coder” features for models deployed to “Simulink Supported Hardware” (Simulink).

Note If you enable this parameter in a model where Simulink Coder is not installed or available in the environment, a question dialog box prompts you to update the model to build without Simulink Coder features.

Category: Hardware Implementation

Settings

 On

Enable the Simulink Coder features.

 Off

Disable the Simulink Coder features.

Indicates that this parameter is enabled. To disable it, first disable the “Use Embedded Coder Features” (Embedded Coder) parameter.

Dependencies

This parameter requires a Simulink Coder or Embedded Coder license.

Command-Line Information

Parameter: `UseSimulinkCoderFeatures`
Value: `'on'` or `'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration”

Comment style

Description

Specify comment style in the generated C/C++ code.

Category: Code Generation > Comments

Settings

Default: `Auto`

`Auto`

For C code, generate single- or multiple-line comments delimited by `/*` and `*/`. For C ++ code, generate single-line comments preceded by `//`.

`Multi-line`

Generate single- or multiple-line comments delimited by `/*` and `*/`.

Example of code generated by using the multiline comment style is:

```
/* Sum: '<Root>/Sum' incorporates:
 *  Constant: '<Root>/INC'
 *  UnitDelay: '<Root>/X'
 */
rtDW.X_g++;
```

`Single-line`

Generate single-line comments preceded by `//`.

Example of code generated by using the single-line comment style is:

```
// Sum: '<Root>/Sum' incorporates:
//   Constant: '<Root>/INC'
//   UnitDelay: '<Root>/X'

rtDW.X_g++;
```

Note For C code generation, select `Single-line` only if your compiler supports it.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires Embedded Coder when generating code.

Command-Line Information

Parameter: `CommentStyle`
Type: character vector
Value: `Auto` | `Multi-line` | `Single-line`
Default: `Auto`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2

Implement each data store block as a unique access point

Description

Specifies whether the generated code contains a single unique variable to hold the value for every Data Store Read and Write operation performed on a Data Store Memory block.

Category: Code Generation > Interface

Settings

Default: off

 On

Creates a unique variable for each Data Store Memory block read/write operation. This unique variable enhances data coherency.

 Off

Does not allocate a unique variable in the generated code for each Data Store Memory block read/write operation. This absence of an unique variable, diminishes data coherency.

Dependencies

This parameter requires Embedded Coder license.

Command-Line Information

Parameter: `DSAsUniqueAccess`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Improve Data Coherency in Generated Code” (Embedded Coder)

Generate separate internal data per entry-point function

Description

Generate a model block signals (block I/O) and discrete states (DWork) acting at the same rate into the same data structure. Depending on how many rates a model has, these structures contain the prefixes `FuncInternalData0`, `FunctionInteralData1`, and so on.

Category: Code Generation > Interface

Settings

Default: off

 On

Store global block signal data (block I/O) and global state data (DWork vectors) operating at the same rate in one data structure in the generated code.

 Off

Do not store global block signal data (block I/O) and global state data (DWork vectors) operating at the same rate in one data structure in the generated code

Tips

Setting this parameter to `On` improves cache performance when deploying a model to a multicore hardware environment that meets these requirements:

- The model has multiple rates and has the **Treat each discrete rate as a separate task** parameter set to `on`.
- The model contains multiple exported functions that run at different rates.

The previous models have separate entry-point functions that different cores can call. A core has its own data cache. Placing data for a single entry-point function in the same core data cache improves execution efficiency because the cache accesses are contiguous rather than spread out over multiple cores.

Example

For a model that generates this code:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T RTBS2F;                       /* '<Root>/RTBS2F' */
  real_T UDS;                          /* '<Root>/UDS' */
  real_T Sum3;                         /* '<Root>/Sum3' */
  real_T Sum1;                         /* '<Root>/Sum1' */
  real_T UDF_DSTATE;                   /* '<Root>/UDF' */
  real_T UDS_DSTATE;                   /* '<Root>/UDS' */
  real_T RTBS2F_Buffer0;               /* '<Root>/RTBS2F' */
  real_T MIXEDDSM;                     /* '<Root>/DSMM' */
  real_T SLOWDSM;                      /* '<Root>/DSMS' */
} DW_demo1_T;
```

If you select **Generate separate internal data per entry-point function**, the generated code now looks like this code:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T RTBS2F_Buffer0;               /* '<Root>/RTBS2F' */
  real_T MIXEDDSM;                     /* '<Root>/DSMM' */
} DW_demo1_T;

/* Internal Data Grouped For Same Function, for system '<Root>' */
typedef struct {
  real_T RTBS2F;                       /* '<Root>/RTBS2F' */
  real_T Sum3;                         /* '<Root>/Sum3' */
  real_T UDF_DSTATE;                   /* '<Root>/UDF' */
} FuncInternalData0_demo1_T;

/* Internal Data Grouped For Same Function, for system '<Root>' */
typedef struct {
  real_T UDS;                          /* '<Root>/UDS' */
  real_T Sum1;                         /* '<Root>/Sum1' */
  real_T UDS_DSTATE;                   /* '<Root>/UDS' */
  real_T SLOWDSM;                      /* '<Root>/DSMS' */
} FuncInternalData1_demo1_T;
```

Dependencies

- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by selecting the **Combine signal/state structures** parameter.

Command-Line Information

Parameter: `CombineSignalStateStructs`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

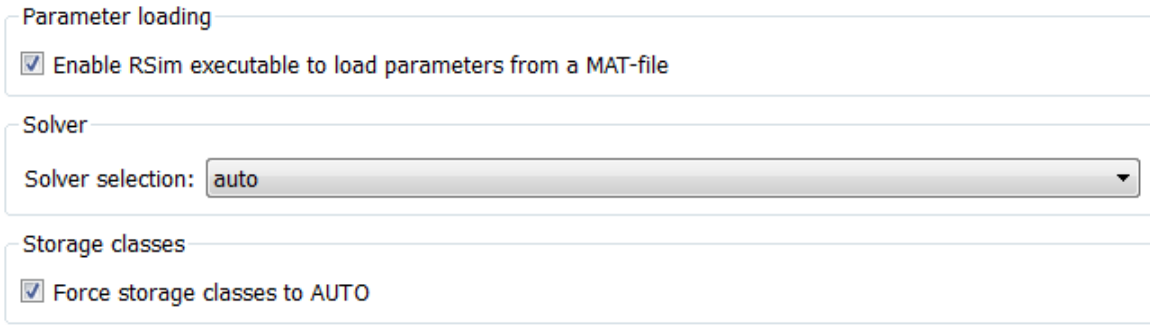
Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Multicore Processor Targets” (Simulink)
- “Time-Based Scheduling” (Embedded Coder)

Configuration Parameters for Simulink Models

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.



In this section...

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 4-7.

See Also

- “Configure and Build Model for Rapid Simulation”
- “Run Rapid Simulations”
- “Code Generation Pane: RSim Target” on page 11-2

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“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 code generator choose the solver. The code generator uses the Simulink solver module if you specify a variable-step solver on the Solver pane. Otherwise, the code generator uses a built-in solver.

`Use Simulink solver module`

Instructs the code generator to use the variable-step solver that you specify on the **Solver** pane.

`Use fixed-step solvers`

Instructs the code generator to use the fixed-step solver that you specify on the **Solver** pane.

Command-Line Information
Parameter: `RSIM_SOLVER_SELECTION`
Type: character vector
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

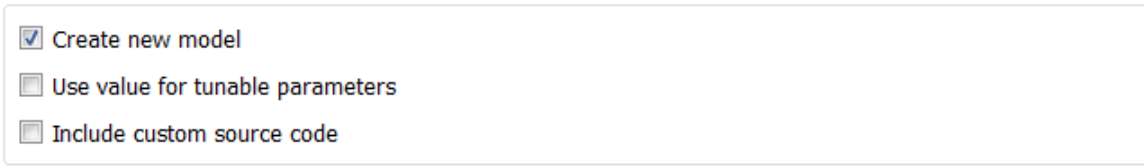
Parameter: `RSIM_STORAGE_CLASS_AUTO`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

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.



In this section...

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 4-7.

See Also

- “Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target”
- “Code Generation Pane: S-Function Target” on page 11-6

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

"Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target"

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target"

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.

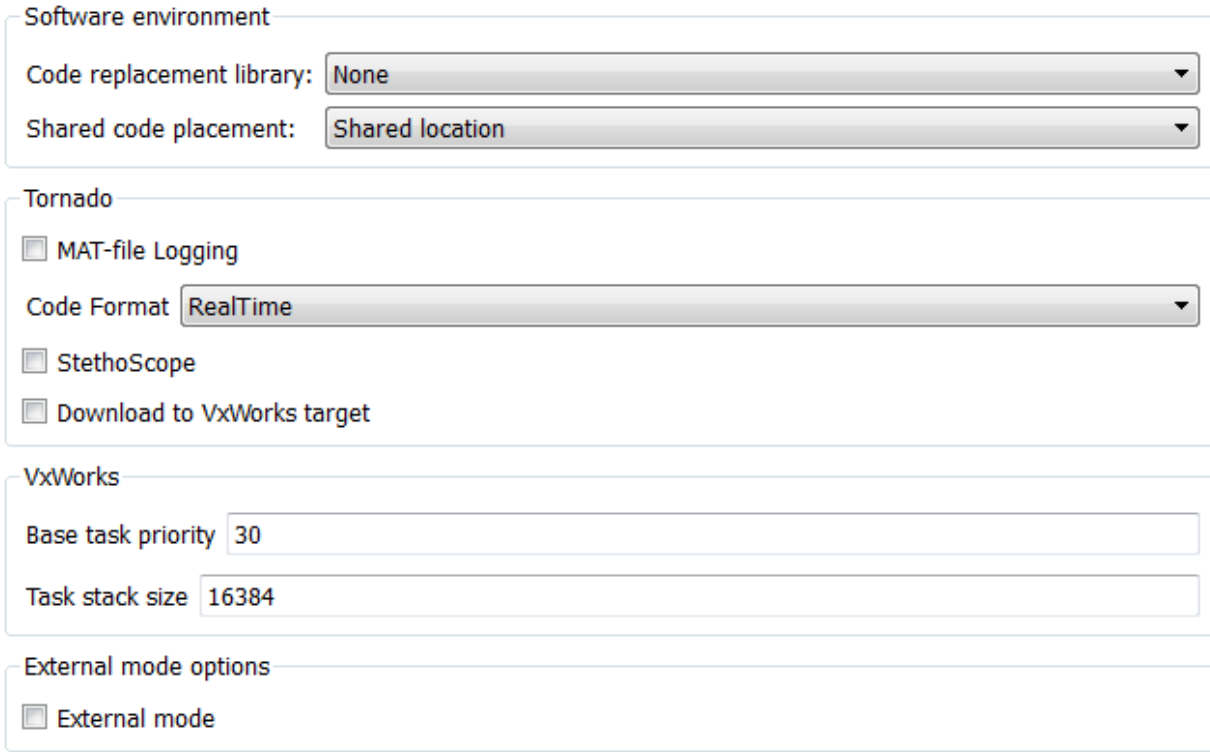
Command-Line Information
Parameter: `AlwaysIncludeCustomSrc`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target"

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.



In this section...

Code Generation: Tornado Target Tab Overview

Control generated code for the Tornado target.

Configuration

This tab appears only if you specify `tornado.tlc` as the “System target file” on page 4-7.

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 11-9

Standard math library

Specify a standard math library for your model.

Settings

Default: `C99 (ISO)`

`C89/C90 (ANSI)`

Generates calls to the ISO/IEC 9899:1990 C standard math library.

`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.

Tips

- The build process checks whether the specified standard math library and toolchain are compatible. If they are not compatible, a warning occurs during code generation and the build process continues.
- When you change the value of the **Language** parameter, the standard math library updates to ISO/IEC 9899:1999 C (`C99 (ISO)`) for C and ISO/IEC 14882:2003 C++ (`C++03 (ISO)`) for C++.

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: character vector
Value: `'C89/C90 (ANSI)'` | `'C99 (ISO)'` | `'C++03 (ISO)'`
Default: `'C99 (ISO)'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

“Run-Time Environment Configuration”

Code replacement library

Specify a code replacement library the code generator uses when producing code for a model.

Settings

Default: `None`

`None`

Does not use a code replacement library.

`Named code replacement library`

Generates calls to a specific platform, compiler, or standards code replacement library. The list of named libraries depends on:

- Installed support packages.
- System target file, language, standard math library, and device vendor configuration.
- Whether you created and registered code replacement libraries, using the Embedded Coder product.

For more information about selections for this parameter, see “Code replacement library” on page 9-11.

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: character vector
Value: `'None'|'GNU C99 extensions'|'Intel IPP for x86-64 (Windows)'|'Intel IPP/SSE for x86-64 (Windows)'|'Intel IPP for x86-64 (Windows for MinGW compiler)'|'Intel IPP/SSE for x86-64 (Windows for MinGW compiler)'|'Intel IPP for x86/Pentium (Windows)'|'Intel IPP/SSE x86/Pentium (Windows)'|'Intel IPP for x86-64 (Linux)'|'Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)'`

Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Run-Time Environment Configuration"

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: character vector
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

- “Run-Time Environment Configuration”
- “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 (see “Model Configuration Parameters: Data Import/Export” (Simulink))
- To Workspace blocks
- Scope blocks with the **Log data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in “Export Simulation Data” (Simulink) 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 support file-scoped data, for example, data items to which you apply the built-in custom storage class `FileScope`.

MAT-file logging does not work in a referenced model, and code is not generated to implement it.

Command-Line Information

Parameter: `MatFileLogging`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

- “Log Program Execution Results”

- “Log Data for Analysis”
- “Virtualized Output Ports Optimization” (Embedded Coder)

MAT-file variable name modifier

Select the text to add to the MAT-file variable names.

Settings

Default: `rt_`

`rt_`
: Adds prefix text.

`_rt`
: Adds suffix text.

`none`
: Does not add text.

Dependency

If you have an Embedded Coder license, this parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: `LogVarNameModifier`
Type: character vector
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

- “Log Program Execution Results”
- “Log Data for Analysis”

Code Format

Specify the code format (generated code features).

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: character vector
Value: `'RealTime'` | `'RealTimeMalloc'`
Default: `'RealTime'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Compare System Target File Support Across Products”

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
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 downloaded 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: character vector
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 code generator 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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Host-Target Communication with External Mode Simulation”

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

“External Mode Simulation with TCP/IP or Serial 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`)

Dependency

This parameter is enabled by the **External mode** check box.

Command-Line Information

Parameter: `ExtModeMexArgs`
Type: character vector
Value: valid arguments
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “External Mode Simulation with TCP/IP or Serial Communication”
- “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: character vector
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

“Control Memory Allocation for Communication Buffers in Target”

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

Application	Setting
Safety precaution	No impact

See Also

“Control Memory Allocation for Communication Buffers in Target”

Recommended Settings Summary for Model Configuration Parameters

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 "Recommended Settings Summary for Model Configuration Parameters" (Embedded Coder).

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	
compexcep="155"" Start time" (Simulink)	No impact	No impact	No impact	0.0	0.0 seconds
"Stop time" (Simulink)	No impact	No impact	No impact	A positive value	10.0 seconds
"Type" (Simulink)	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Variable-step` (you must change to `Fixed-step` for code generation)
"Solver" (Simulink)	No impact	No impact	No impact	`Discrete (no continuous states)`	`ode3 (Bogacki-Shampine)`
"Periodic sample time constraint" (Simulink)	No impact	No impact	No impact	`Specified` or `Ensure sample time independent`	`Unconstrained`
"Sample time properties" (Simulink)	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	`''`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
“Treat each discrete rate as a separate task” (Simulink)	No impact	No impact	No impact	No impact	On
“Automatically handle rate transition for data transfer” (Simulink)	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” (Simulink)	No impact	No impact	No recommendation	No recommendation	Off
“Initial state” (Simulink)	No impact	No impact	No recommendation	No recommendation	Off
“Time” (Simulink)	No impact	No impact	No recommendation	No recommendation	On
“States” (Simulink)	No impact	No impact	No recommendation	No recommendation	Off
“Output” (Simulink)	No impact	No impact	No recommendation	No recommendation	On
“Final states” (Simulink)	No impact	No impact	No recommendation	No recommendation	Off
“Signal logging” (Simulink)	No impact	No impact	No recommendation	No recommendation	On
“Record logged workspace data in Simulation Data Inspector” (Simulink)	No impact	No impact	No recommendation	No recommendation	Off
“Limit data points” (Simulink)	No impact	No impact	No recommendation	No recommendation	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
“Decimation” (Simulink)	No impact	No impact	No recommendation	No recommendation	1
“Format” (Simulink)	No impact	No impact	No recommendation	No recommendation	`Array`
“Output options” (Simulink)	No impact	No impact	No recommendation	No recommendation	`Refine output`
“Refine factor” (Simulink)	No impact	No impact	No recommendation	No recommendation	1
“Output times” (Simulink)	No impact	No impact	No recommendation	No recommendation	`'[]'`

Mapping Application Requirements to the Math and Data Types Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Default for underspecified data type" (Simulink)	No impact	No impact	`single`	No impact	`double`
"Use division for fixed-point net slope computation" (Simulink)	No impact	No impact	On (when target hardware supports efficient division) Off (otherwise)	No impact	`off`
"Application lifespan (days)" (Simulink)	No impact	No impact	Finite value	`inf`	`auto`
"Use floating-point multiplication to handle net slope corrections" (Simulink)	No impact	No impact	On (when target hardware supports efficient multiplication) Off (otherwise)	No recommendation	Off
"Remove code from floating-point to integer conversions that wraps out-of-range values" on page 15-14	Off	Off	On (execution, ROM) No impact (RAM)	No recommendation	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
*The command-line value is reverse of the listed value.					

Mapping Application Requirements to the Optimization Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Default parameter behavior" on page 15-18	`Tunable` (GRT) `Inlined` (ERT)	`Inlined`	`Inlined`	No impact	`Tunable` (GRT) `Inlined` (ERT)
"Loop unrolling threshold" on page 15-30	No impact	No impact	>0	No impact	5
"Maximum stack size (bytes)" on page 15-32	No impact	No impact	No impact	No impact	`Inherit from target`
"Use memcpy for vector assignment" on page 15-22	No impact	No impact	On	No impact	On
"Memcpy threshold (bytes)" on page 15-24	No impact	No impact	Accept default or determine target-specific optimal value	No impact	64
"Inline invariant signals" on page 15-20	Off	Off	On	No impact	Off
"Remove code from floating-point to integer conversions that wraps out-of-range values" on page 15-14	Off	Off	On (execution, ROM) No impact (RAM)	No recommendation	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Use bitsets for storing state configuration" on page 15-56	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off
"Use bitsets for storing Boolean data" on page 15-58	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" (Simulink)	`error`	No impact	No impact	`error`	`warning`
"Minimize algebraic loop" (Simulink)	No impact	No impact	No impact	`error`	`warning`
"Block priority violation" (Simulink)	No impact	No impact	No impact	`error`	`warning`
"Consecutive zero-crossings violation" (Simulink)	No impact	No impact	No impact	`warning` or `error`	`error`
"Unspecified inheritability of sample time" (Simulink)	No impact	No impact	No impact	`error`	`warning`
"Solver data inconsistency" (Simulink)	`warning`	No impact	`none`	No impact	`warning`
"Automatic solver parameter selection" (Simulink)	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" (Simulink)	No impact	No impact	No impact	`error`	`none`
"Multitask rate transition" (Simulink)	No impact	No impact	No impact	`error`	`error`
"Single task rate transition" (Simulink)	No impact	No impact	No impact	`none` or `error`	`none`
"Multitask conditionally executed subsystem" (Simulink)	No impact	No impact	No impact	`error`	`error`
"Tasks with equal priority" (Simulink)	No impact	No impact	No impact	`none` or `error`	`warning`
"Enforce sample times specified by Signal Specification blocks" (Simulink)	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” (Simulink)	No impact	No impact	No impact	`Explicit only`	`Explicit only`
“Division by singular matrix” (Simulink)	No impact	No impact	No impact	`error`	`none`
“Underspecified data types” (Simulink)	No impact	No impact	No impact	`error`	`none`
“Simulation range checking” (Simulink)	`warning` or `error`	`warning` or `error`	`none`	`error`	`none`
“Wrap on overflow” (Simulink)	No impact	No impact	No impact	`error`	`warning`
“Saturate on overflow” (Simulink)	No impact	No impact	No impact	`error`	`warning`
“Inf or NaN block output” (Simulink)	No impact	No impact	No impact	`error`	`none`
“"rt" prefix for identifiers” (Simulink)	No impact	No impact	No impact	`error`	`error`
“Detect downcast” (Simulink)	No impact	No impact	No impact	`error`	`error`
“Detect overflow” (Simulink)	No impact	No impact	No impact	`error`	`error`
“Detect underflow” (Simulink)	No impact	No impact	No impact	`error`	`none`
“Detect precision loss” (Simulink)	No impact	No impact	No impact	`error`	`error`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Detect loss of tunability" (Simulink)	No impact	No impact	No impact	`error`	`warning` for GRT-based targets `error` for ERT-based targets
"Detect read before write" (Simulink)	No impact	No impact	No impact	`error`	`Enable all as warnings`
"Detect write after read" (Simulink)	No impact	No impact	No impact	`error`	`Enable all as warning`
"Detect write after write" (Simulink)	No impact	No impact	No impact	`error`	`Enable all as errors`
"Multitask data store" (Simulink)	No impact	No impact	No impact	`error`	`warning`
"Duplicate data store names" (Simulink)	`warning`	No impact	`none`	No impact	`none`
"Check undefined subsystem initial output" (Simulink)	No impact	No impact	No impact	On	On
"Check runtime output of execution context" (Simulink)	No impact	No impact	No impact	On	Off

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" (Simulink)	No impact	No impact	No impact	`warning`	`none`
"Vector/matrix block input conversion" (Simulink)	No impact	No impact	No impact	`error`	`none`
"32-bit integer to single precision float conversion" (Simulink)	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” (Simulink)	No impact	No impact	No impact	`error`	`none`
“Unconnected block input ports” (Simulink)	No impact	No impact	No impact	`error`	`warning`
“Unconnected block output ports” (Simulink)	No impact	No impact	No impact	`error`	`warning`
“Unconnected line” (Simulink)	No impact	No impact	No impact	`error`	`none`
“Unspecified bus object at root Outport block” (Simulink)	No impact	No impact	No impact	`error`	`warning`
“Element name mismatch” (Simulink)	No impact	No impact	No impact	`error`	`warning`
“Bus signal treated as vector” (Simulink)	No impact	No impact	No impact	`error`	`none`
“Invalid function-call connection” (Simulink)	No impact	No impact	No impact	`error`	`error`
“Context-dependent inputs” (Simulink)	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" (Simulink)	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" (Simulink)	No impact	No impact	No impact	No recommendation	`none`
"Port and parameter mismatch" (Simulink)	No impact	No impact	No impact	`error`	`none`
"Invalid root Inport/Outport block connection" (Simulink)	No impact	No impact	No impact	`error`	`none`
"Unsupported data logging" (Simulink)	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" (Simulink)	No impact	No impact	No impact	No impact	`warning`
"Block diagram contains parameterized library links" (Simulink)	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, events, messages, and functions" (Simulink)	`warning`	No impact	No impact (for simulation and during development) `none` (for production code generation)	`warning`	`warning`
"Unexpected backtracking" (Simulink)	`warning`	No impact	No impact	`error`	`warning`
"Invalid input data access in chart initialization" (Simulink)	`warning`	No impact	No impact	`error`	`warning`
"No unconditional default transitions" (Simulink)	`warning`	No impact	No impact (for simulation and during development) `none` (for production code generation)	`error`	`warning`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Transition outside natural parent" (Simulink)	`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" (Simulink)	No impact	No impact	No impact	Select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`None` if specified system target file is `ert.tlc`, `realtime.tlc`, or `autosar.tlc`. Otherwise, `Determine by Code Generation system target file`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Device vendor" (Simulink)	No impact	No impact	No impact	Select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`Intel`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Device type" (Simulink)	No impact	No impact	No impact	Select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`x86–64 (Windows64)`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Number of bits: char" (Simulink)	No impact	No impact	Target specific	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`char` 8, `short` 16, `int` 32, `long` 32, `long long` 64, `float` 32, `double` 64, `native` 32, `pointer` 32

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Largest atomic size: integer" (Simulink)	No impact	No impact	Target specific	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	integer `Char`, floating-point `Float`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Byte ordering" (Simulink)	No impact	No impact	No impact	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`Little Endian`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	**Traceability**	**Efficiency**	**Safety precaution**	
"Signed integer division rounds to" (Simulink)	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 recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	`Zero`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Shift right on a signed integer as arithmetic shift" (Simulink)	No impact	No impact	On	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Support long long" (Simulink)	No impact	No impact	On (execution, ROM)	No recommendation for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	Off

Mapping Application Requirements to the Model Referencing Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Rebuild" (Simulink)	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" (Simulink)	No impact	No impact	No impact	`error if rebuild required`	`error if rebuild required`
"Enable parallel model reference builds" (Simulink)	No impact	No impact	No impact	No impact	Off
"MATLAB worker initialization for builds" (Simulink)	No impact	No impact	No impact	No impact	`None`
"Total number of instances allowed per top model" (Simulink)	No impact	No impact	No impact	No recommendation	`Multiple`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
“Pass fixed-size scalar root inputs by value for code generation” (Simulink)	No impact	No impact	No impact	No recommendation	Off
“Minimize algebraic loop occurrences” (Simulink)	No impact	No impact	No impact	No recommendation	Off
“Propagate sizes of variable-size signals” (Simulink)	No impact	No impact	No impact	No recommendation	`Infer from blocks in model`
“Model dependencies” (Simulink)	No impact	No impact	No impact	No recommendation	`''`

Mapping Application Requirements to the Simulation Target Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Ensure memory integrity" (Simulink)	On	No impact	No recommendation	On	On
"Echo expressions without semicolons" (Simulink)	On	No impact	Off	No impact	On
"Ensure responsiveness" (Simulink)	On	No recommendation	No recommendation	No recommendation	On
"Simulation target build mode" (Simulink)	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" (Simulink)	No impact	No impact	No impact	No recommendation	`{}`

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
“Import custom code” (Simulink)	On	No impact	No impact	On	On
“Source file” (Simulink)	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Header file” (Simulink)	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Initialize function” (Simulink)	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Terminate function” (Simulink)	No recommendation	No recommendation	No recommendation	No recommendation	' '
“Include directories” (Simulink)	No impact	No impact	No impact	No recommendation	' '
“Source files” (Simulink)	No impact	No impact	No impact	No recommendation	' '
“Libraries” (Simulink)	No impact	No impact	No impact	No recommendation	' '
“Defines” (Simulink)	No impact	No impact	No impact	No recommendation	' '

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 on page 4-7	No impact	No impact	No impact	No impact (GRT) ERT based (ERT)	`grt.tlc`
“Language” on page 4-10	No impact	No impact	No impact	No impact	C
compexcep="155" “Compiler optimization level” on page 4-20	`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” on page 4-22	`Optimizations off (faster builds)`	`Optimizations off (faster builds)`	`Optimizations on (faster runs)`	No impact	`Optimizations off (faster builds)`
“Generate makefile” on page 4-24	No impact	No impact	No impact	No impact	On
“Make command” on page 4-26	No impact	No impact	No impact	No recommendation	`make_rtw`
“Template makefile” on page 4-28	No impact	No impact	No impact	No impact	`grt_default_tmf`
“Select objective / Prioritized objectives” on page 4-30	`Debugging`	Not applicable for GRT-based targets	`Execution efficiency`	No recommendation	`Unspecified`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Check model before generating code" on page 4-38	`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" on page 4-40	Off	No impact	No impact	No impact	Off
"Verbose build" on page 10-55	On	No impact	No impact	No recommendation	On
"Retain .rtw file" on page 10-57	On	No impact	No impact	No impact	Off
"Profile TLC" on page 10-59	On	No impact	No impact	No impact	Off
"Start TLC debugger when generating code" on page 10-61	On	No impact	No impact	No impact	Off
"Start TLC coverage when generating code" on page 10-63	On	No impact	No impact	No impact	Off
"Enable TLC assertion" on page 10-65	On	No impact	No impact	No recommendation	Off

Mapping Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Create code generation report" on page 5-5	On	On	No impact	No recommendation	Off
"Open report automatically" on page 5-8	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 page 6-5	On	On	No impact	No recommendation	On
"Simulink block comments" on page 6-7	On	On	No impact	No recommendation	On
"Stateflow object comments" on page 6-11	On	On	No impact	No recommendation	Off
"Show eliminated blocks" on page 6-15	On	On	No impact	No recommendation	On
"Verbose comments for 'Model default' storage class" on page 6-17	On	On	No impact	No recommendation	On
"Operator annotations" on page 6-19	No impact	On	No impact	No recommendation	On

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" on page 7-33	Valid value	>30	No impact	>30	31
"Use the same reserved names as Simulation Target" on page 7-56	No impact	No impact	No impact	No impact	Off
"Reserved names" on page 7-58	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" on page 8-5	No impact	No impact	No impact	No impact	Off
"Source file" on page 8-7	No impact	No impact	No impact	No impact	' '
"Header file" on page 8-9	No impact	No impact	No impact	No impact	' '
"Initialize function" on page 8-11	No impact	No impact	No impact	No impact	' '
"Terminate function" on page 8-13	No impact	No impact	No impact	No impact	' '
"Include directories" on page 8-15	No impact	No impact	No impact	No impact	' '
"Source files" on page 8-17	No impact	No impact	No impact	No impact	' '
"Libraries" on page 8-19	No impact	No impact	No impact	No impact	' '
"Defines" on page 8-21	No impact	No impact	No impact	No impact	' '

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" on page 10-21	No impact	No impact	Valid library	No impact	`C99 (ISO)`
"Code replacement library" on page 9-11	No impact	No impact	Valid library	No impact	`None`
"Shared code placement" on page 9-14	`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" on page 9-18	No impact	No impact	Off (Execution, ROM) No impact (RAM)	Norecommendation	On
"Code interface packaging" on page 9-29	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" on page 9-33	`Warning` or `Error`	No impact	`None`	No impact	`Error`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Classic call interface" on page 10-30	No impact	Off	Off (execution, ROM), No impact (RAM)	No recommendation	Off (except On for GRT models created before R2012a)
"Single output/ update function" on page 10-36	On	On	On	No recommendation	On
"MAT-file logging" on page 10-50	On	No impact	Off	Off	On (GRT) Off (ERT)
"MAT-file variable name modifier" on page 10-53	No impact	No impact	No impact	No impact	`rt_`
"Generate C API for: signals" on page 9-46	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Generate C API for: parameters" on page 9-48	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Generate C API for: states" on page 9-50	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Generate C API for: root-level I/O" on page 9-52	No impact	No impact	No impact	No impact (development) Off (production)	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"ASAP2 interface" on page 9-54	No impact	No impact	No impact	No impact (development) Off (production)	Off
"External mode" on page 9-56	No impact	No impact	No impact	No impact (development) Off (production)	Off
"Transport layer" on page 9-58	No impact	No impact	No impact	No impact	`tcpip`
"MEX-file arguments" on page 9-61	No impact	No impact	No impact	No impact	`''`
"Static memory allocation" on page 9-64	No impact	No impact	No impact	No impact	Off
"Static memory buffer size" on page 11-26	No impact	No impact	No impact	No impact	1000000

Model Advisor Checks

Simulink Coder Checks

In this section...

Simulink Coder Checks Overview

Use Simulink Coder Model Advisor checks to configure your model for code generation.

See Also

- "Run Model Checks" (Simulink)
- "Simulink Checks" (Simulink)
- "Embedded Coder Checks" (Embedded Coder)

Identify blocks using one-based indexing

Check ID: `mathworks.codegen.cgsl_0101`

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.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “cgsl_0101: Zero-based indexing” (Simulink).
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check solver for code generation

Check ID: `mathworks.codegen.SolverCodeGen`

Check model solver and sample time configuration settings.

Description

Incorrect configuration settings can stop the code generator from producing 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 **Type** (Simulink) to `Fixed-step`.
Multitasking diagnostic options are not set to `error`.	In the Configuration Parameters dialog box, on the **Diagnostics** pane, set • **Sample Time > Multitask conditionally executed subsystem** (Simulink) to `error` • **Sample Time > Multitask rate transition** (Simulink) to `error` • **Data Validity > Multitask data store** (Simulink) to `error`

Tips

You do not have to modify the solver settings to generate code from a subsystem. The 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 Time-Based Scheduling”
- “Execute Multitasking Models”

Check for blocks not supported by code generation

Check ID: `mathworks.codegen.codeGenSupport`

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.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Blocks and Products Supported for C Code Generation”
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check and update model to use toolchain approach to build generated code

Check ID: `mathworks.codegen.toolchainInfoUpgradeAdvisor.check`

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 4-13
- “Build configuration” on page 4-15

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 `ert.tlc`, `ert_shrlib.tlc`, or `grt.tlc`.

Condition	Recommended Action	Comment
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 `Optimizations on (faster runs)`. Clicking **Update Model** sets **Compiler Optimization Level** to its default value, `Optimizations off (faster builds)`, and then upgrades the model. The upgraded model has **Build Configuration** set to `Faster Builds`. 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: • **Generate makefile** (default: Enabled) • **Template makefile** (default: Target-specific default TMF) • **Compiler optimization level** (default: `Optimizations off (faster builds)`) • **Make command** (default: `make_rtw` 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 embedded target model to use ert.tlc system target file

Check ID: `mathworks.codegen.codertarget.check`

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 `ert.tlc - Embedded Coder`.	No action
System target file is set to `idelink_ert.tlc` or `idelink_grt.tlc` 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

“Toolchain Configuration”

Check and update models that are using targets that have changed significantly across different releases of MATLAB

Check ID:
`mathworks.codegen.realtime2CoderTargetInfoUpgradeAdvisor.check`

Check and update models with Simulink targets that have changed significantly across different releases of MATLAB.

Description

Save a model that you have updated to work with the current installation of MATLAB.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
Model uses a target that has changed significantly since the release of MATLAB in which it was originally saved.	Save model
Model does not use a Simulink target or is using the latest version of the target.	No action
Model is automatically updated.	Save model

Condition	Recommended Action
Invalid external mode configuration.	In the **Configuration Parameters > Interface** pane, update the external mode parameter settings to match characteristics of your host-target connection.
Model is corrupted.	Close and reopen the model. If the issue persists, reset **Configuration Parameters > Hardware Implementation > Hardware board**.

Action Results

Clicking **Save model** updates the model to work with the current installation of MATLAB and saves the model.

See Also

“Configure Production and Test Hardware”

Check for blocks that have constraints on tunable parameters

Check ID: `mathworks.codegen.ConstraintsTunableParam`

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 incorrect 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: • 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: • 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 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**.

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- 1-D Lookup Table
- 2-D Lookup Table
- “What Is a Model Advisor Exclusion?” (Simulink Check)

Check for model reference configuration mismatch

Check ID: `mathworks.codegen.MdlrefConfigMismatch`

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 Reference Basics” (Simulink)
- “Set Configuration Parameters for Model Referencing” (Simulink)

Check sample times and tasking mode

Check ID: `mathworks.codegen.SampleTimesTaskingMode`

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 **"Treat each discrete rate as a separate task" (Simulink)** 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, clear the checkbox for the **"Treat each discrete rate as a separate task" (Simulink)** parameter, or change the settings on the **Hardware Implementation** pane.

See Also

"Time-Based Scheduling and Code Generation"

Check for code generation identifier formats used for model reference

Check ID: `mathworks.codegen.ModelRefRTWConfigCompliance`

Checks for referenced models in a model referencing hierarchy for which code generation changes configuration parameter settings that involve identifier formats.

Description

In referenced models, if the following **Configuration Parameters > Code Generation > Symbols** parameters have settings that do not contain a `$R` token (which represents the name of the reference model), code generation prepends the `$R` token to the identifier format.

- **Global variables**
- **Global types**
- **Subsystem methods**
- **Constant macros**

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
A script that operates on generated code uses model names that code generation changes.	Update the script to use the generated name (which includes an appended $R token).

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 from Simulink, Simulink Coder, and Embedded Coder for each of the code generation objectives. Two checks unique to the Code Generation Advisor are included below the list.

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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
"Check model configuration settings against code generation objectives" on page 12-25	Included	Included	Included	Included	Included	Included	Included (see **Note** below)	Included
"Check for optimal bus virtuality" (Simulink)	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
"Identify questionable blocks within the specified system" on page 12-24	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
"Check the hardware implementation" (Embedded Coder)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify questionable software environment specifications” (Embedded Coder)	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)” (Embedded Coder)	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” (Embedded Coder)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify blocks that generate expensive rounding code” (Embedded Coder)	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” (Embedded Coder)	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 12-2	Included	Included	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify lookup table blocks that generate expensive out-of-range checking code” (Embedded Coder)	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 output types of logic blocks” (Embedded Coder)	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” (Simulink)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues” (Simulink)	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Identify block output signals with continuous sample time and non-floating point data type” (Simulink)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for blocks that have constraints on tunable parameters” on page 12-10	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” (Simulink)	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check structure parameter usage with bus signals” (Simulink)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check data store block sample times for modeling errors” (Simulink)	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for potential ordering issues involving data store access” (Simulink)	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for blocks not recommended for C/C++ production code deployment” (Embedded Coder)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for blocks not recommended for MISRA C: 2012” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for unsupported block names” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check usage of Assignment blocks” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for bitwise operations on signed integers” (Embedded Coder)	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:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for recursive function calls” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for equality and inequality operations on floating-point values” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for switch case expressions without a default case” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A

Note When the Code Generation Advisor checks your model against the MISRA C:2012 guidelines objective, the tool does not consider all of the configuration parameter settings that are checked by the MISRA C:2012 guidelines checks in the Model Advisor. For a

complete check of configuration parameter settings, run the checks under the **By Task > Modeling Guidelines for MISRA C:2012** node in the Model Advisor.

See Also

- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Run Model Checks” (Simulink)
- “Simulink Checks” (Simulink)
- “Simulink Coder Checks” on page 12-2
- “Simulink Check Checks” (Simulink Check)

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 code generator.	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.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

"Blocks and Products Supported for C Code Generation"

"What Is a Model Advisor Exclusion?" (Simulink Check)

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. **Note** A change to one parameter value can impact other parameters. Passing the check might take multiple iterations.

Action Results

Clicking **Modify Parameters** changes the parameter values to the recommended values.

See Also

- "Recommended Settings Summary for Model Configuration Parameters" (Embedded Coder)

- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)

Code Generation Advisor Checks

In this section...

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 from Simulink, Simulink Coder, and Embedded Coder for each of the code generation objectives. Two checks unique to the Code Generation Advisor are included below the list.

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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check model configuration settings against code generation objectives ” on page 12-37	Included	Included	Included	Included	Included	Included	Included (see **Note** below)	Included

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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for optimal bus virtuality” (Simulink)	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
“Identify questionable blocks within the specified system” on page 12-36	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
“Check the hardware implementation” (Embedded Coder)	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” (Embedded Coder)	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

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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify questionable code instrumentation (data I/O)” (Embedded Coder)	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” (Embedded Coder)	N/A	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A
“Identify blocks that generate expensive rounding code” (Embedded Coder)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify questionable fixed-point operations” (Embedded Coder)	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 12-2	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” (Embedded Coder)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check output types of logic blocks” (Embedded Coder)	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” (Simulink)	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” (Simulink)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Identify block output signals with continuous sample time and non-floating point data type” (Simulink)	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 12-10	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” (Simulink)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check structure parameter usage with bus signals” (Simulink)	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check data store block sample times for modeling errors” (Simulink)	N/A	N/A	N/A	Included	N/A	N/A	N/A	N/A
“Check for potential ordering issues involving data store access” (Simulink)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for blocks not recommended for C/C++ production code deployment” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for blocks not recommended for MISRA C: 2012” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for unsupported block names” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check usage of Assignment blocks” (Embedded Coder)	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: 2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for bitwise operations on signed integers” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for recursive function calls” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A
“Check for equality and inequality operations on floating-point values” (Embedded Coder)	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:2012 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
“Check for switch case expressions without a default case” (Embedded Coder)	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedded Coder is available	N/A

Note When the Code Generation Advisor checks your model against the MISRA C:2012 guidelines objective, the tool does not consider all of the configuration parameter settings that are checked by the MISRA C:2012 guidelines checks in the Model Advisor. For a complete check of configuration parameter settings, run the checks under the **By Task > Modeling Guidelines for MISRA C:2012** node in the Model Advisor.

See Also

- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)
- “Run Model Checks” (Simulink)
- “Simulink Checks” (Simulink)
- “Simulink Coder Checks” on page 12-2
- “Simulink Check Checks” (Simulink Check)

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 code generator.	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.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

“Blocks and Products Supported for C Code Generation”

“What Is a Model Advisor Exclusion?” (Simulink Check)

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. **Note** A change to one parameter value can impact other parameters. Passing the check might take multiple iterations.

Action Results

Clicking **Modify Parameters** changes the parameter values to the recommended values.

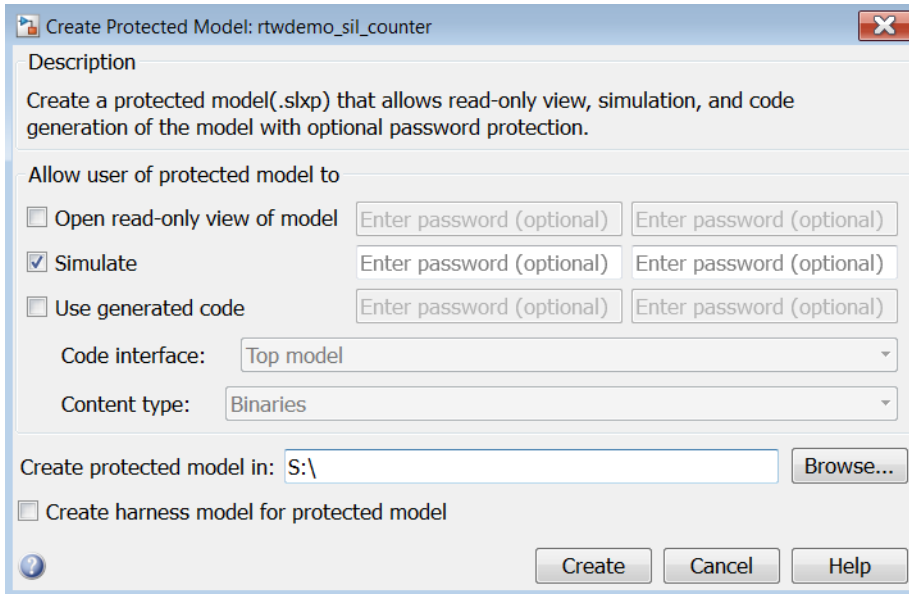
See Also

- “Recommended Settings Summary for Model Configuration Parameters” (Embedded Coder)
- “Application Objectives Using Code Generation Advisor”
- “Configure Model for Code Generation Objectives by Using Code Generation Advisor” (Embedded Coder)

13

Parameters for Creating Protected Models

Create Protected Model



In this section...

Create Protected Model: Overview

Create a protected model (.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.

To open the Create Protected Model dialog box, right-click the model block that references the model for which you want to generate protected model code. From the context menu, select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**.

See Also

- “Reference Protected Models from Third Parties” (Simulink)
- “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.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Create a Protected Model”
- “Protect Models for Third-Party Use”

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 Models for Third-Party Use”

Use generated code

Allows user to generate code for the protected model with optional password protection. Selecting **Use generated code**:

- Enables Simulation Report and Code Generation Report for the protected model.
- Enables code generation.
- Enables support for simulation.

Settings

Default: Off

 On

User can generate code for the protected model. For password protection, create and verify a password with a minimum of four characters.

 Off

User cannot generate code for the protected model.

Dependencies

- To generate code, you must also select the **Simulate** check box.
- This parameter enables **Code interface** and **Content type**.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Support in Protected Models”
- “Protect Models for Third-Party Use”

Code interface

Specify the interface for the generated code.

Settings

Default: `Model reference`

`Model reference`

Specifies the model reference interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations to verify code.

`Top model`

Specifies the standalone interface. Users of the protected model can run Model block SIL or PIL simulations to verify the protected model code.

Dependencies

- Requires an Embedded Coder license
- This parameter is enabled if you:
  - Specify an ERT (`ert.tlc`) system target file.
  - Select the **Use generated code** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Support in Protected Models”
- “Protect Models for Third-Party Use”
- “Code Interfaces for SIL and PIL” (Embedded Coder)

Content type

Select the appearance of the generated code.

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 **Use generated code** check box.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Code Generation Support in Protected Models”
- “Protect Models for Third-Party Use”

Create protected model in

Specify the folder path for the protected model.

Settings

Default: Current working folder

Dependencies

A model that you protect must be available on the MATLAB path.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Protect Models for Third-Party Use”
- “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”

14

Tools in Simulink Coder—Alphabetical List

Code Replacement Viewer

Explore content of code replacement libraries

Description

The Code Replacement Viewer displays the content of code replacement libraries and tables. You can use this tool to explore and choose a code replacement library or to view a predefined code replacement table. If you develop a custom code replacement library, you can use this viewer to verify table entries for the following properties:

- Argument order is correct.
- Conceptual argument names match code generator naming conventions.
- Implementation argument names are correct.
- Header or source file specification is not missing.
- I/O types are correct.
- Relative priority of entries is correct (highest priority is 0, and lowest priority is 100).
- Saturation or rounding mode specifications are not missing.

If you specify a library name when you open the viewer, the viewer displays the code replacement tables for that library. If you specify a table name when you open the viewer, the viewer displays the function and operator code replacement entries for that table. The viewer can only display code replacement tables that are defined. For more information on creating code replacement tables, see “Define Code Replacement Mappings” (Embedded Coder).

Abbreviated Entry Information

In the middle pane, the viewer displays entries that are in the selected code replacement table, along with abbreviated information for each entry.

Field	Description
Name	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`).

Field	Description
Implementation	Name of the implementation function, which can match or differ from **Name**.
NumIn	Number of input arguments.
In1Type	Data type of the first conceptual input argument.
In2Type	Data type of the second conceptual input argument.
OutType	Data type of the conceptual output argument.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
UsageCount	Not used.

Detailed Entry Information

In the middle pane, when you select an entry, the viewer displays entry details.

Field	Description
Description	Text description of the table entry (can be empty).
Key	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`), and the number of conceptual input arguments.
Implementation	Name of the implementation function, and the number of implementation input arguments.
Implementation type	Type of implementation: `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro.

Saturation mode	Saturation mode that the implementation function supports. One of:
	`RTW_SATURATE_ON_OVERFLOW`
	`RTW_WRAP_ON_OVERFLOW`
	`RTW_SATURATE_UNSPECIFIED`

Field	Description

Rounding modes	Rounding modes that the implementation function supports. One or more of:
	`RTW_ROUND_FLOOR`
	`RTW_ROUND_CEILING`
	`RTW_ROUND_ZERO`
	`RTW_ROUND_NEAREST`
	`RTW_ROUND_NEAREST_ML`
	`RTW_ROUND_SIMPLEST`
	`RTW_ROUND_CONV`
	`RTW_ROUND_UNSPECIFIED`

GenCallback file	Not used.
Implementation header	Name of the header file that declares the implementation function.
Implementation source	Name of the implementation source file.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
Total Usage Count	Not used.
Entry class	Class from which the current table entry is instantiated.
Conceptual arguments	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), and data type for each conceptual argument.
Implementation	Name, I/O type (`RTW_IO_OUTPUT` or `RTW_IO_INPUT`), data type, and alignment requirement for each implementation argument.

Fixed-Point Entry Information

When you select an operator entry that specifies net slope fixed-point parameters, the viewer displays fixed-point information.

Field	Description
Net slope adjustment factor F	Slope adjustment factor (F) part of the net slope factor, $F2^E$, for net slope table entries. You use this factor with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Net fixed exponent E	Fixed exponent (E) part of the net slope factor, $F2^E$, for net slope table entries. You use this fixed exponent with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Slopes must be the same	Indicates whether code replacement request processing must check that the slopes on arguments (input and output) are equal. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.
Must have zero net bias	Indicates whether code replacement request processing must check that the net bias on arguments is zero. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.

Open the Code Replacement Viewer

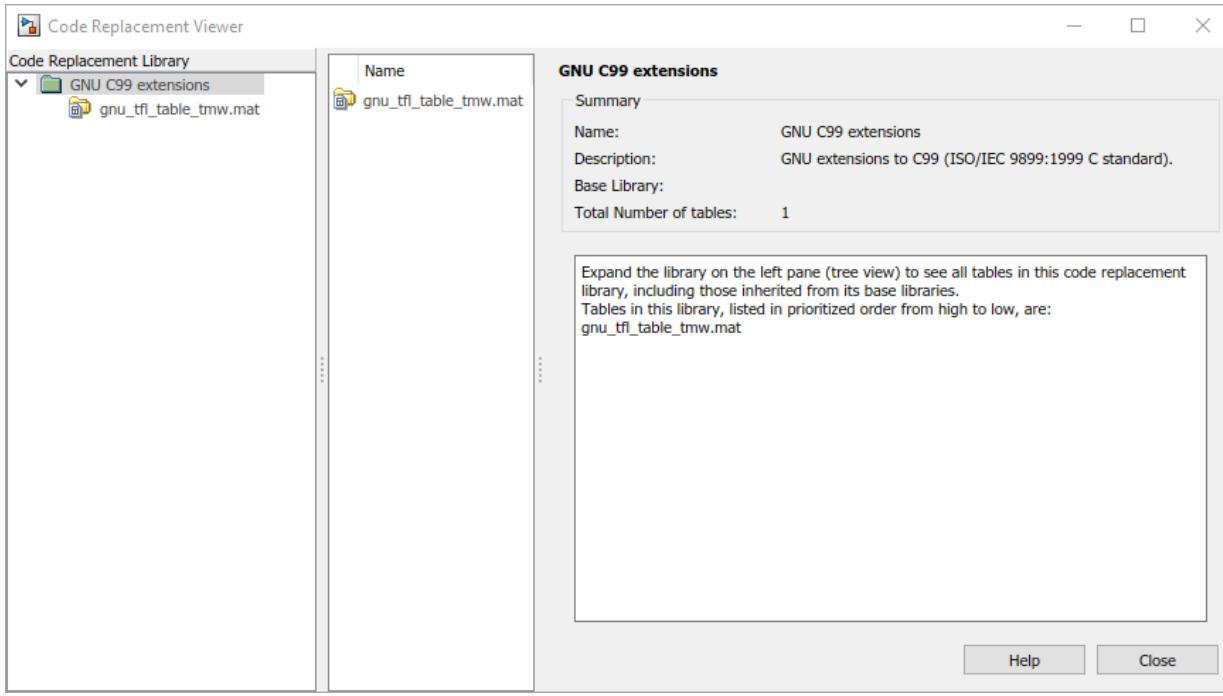
Open from the MATLAB command prompt using `crviewer`.

Examples

Display Contents of Code Replacement Library

This example opens the registered code replacement library `GNU C99 extensions`.

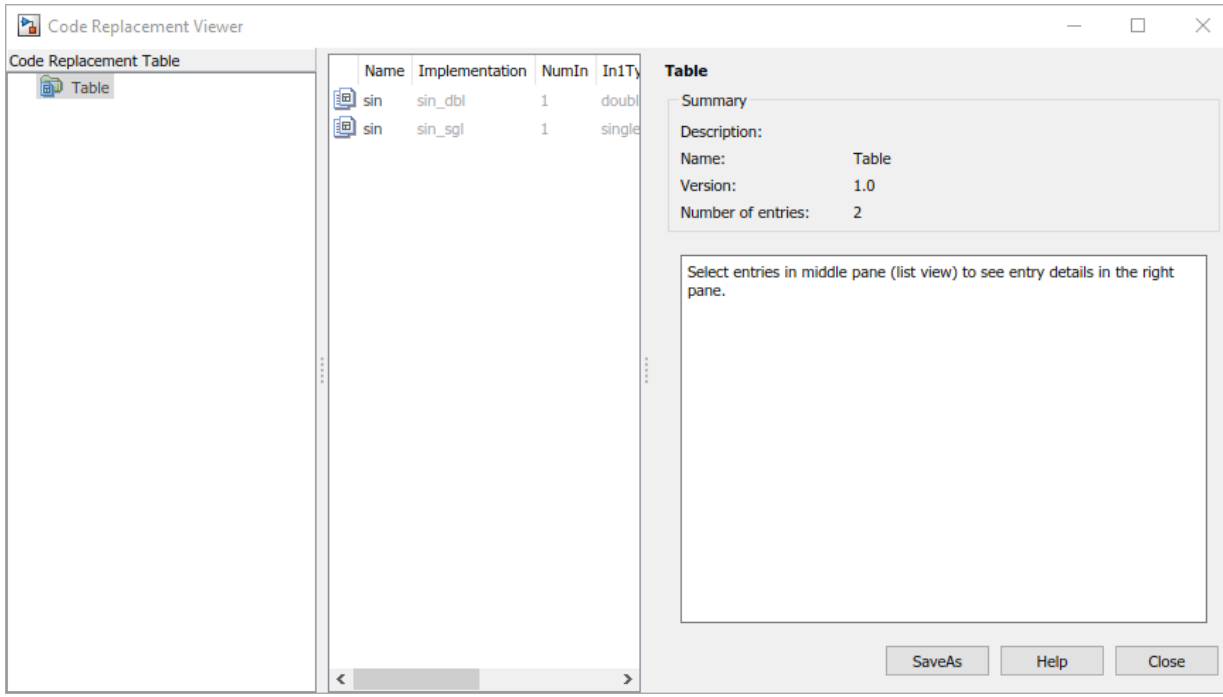
```
crviewer('GNU C99 extensions')
```



Display Contents of Code Replacement Table

This example opens a predefined code replacement table `crl_table_sinfcn`. To learn how to create this example table, see “Define Code Replacement Mappings” (Embedded Coder).

```
crviewer(crl_table_sinfcn)
```



- “Choose a Code Replacement Library”

Programmatic Use

`crviewer('library')` opens the Code Replacement Viewer and displays the contents of `library`, where `library` is a character vector that names a registered code replacement library.

`crviewer(table)` opens the Code Replacement Viewer and displays the contents of a predefined `table`, where `table` is a MATLAB file that defines code replacement tables. The table must be user predefined and the file must be in the current folder or on the MATLAB path.

See Also

Topics

“Choose a Code Replacement Library”
“What Is Code Replacement?”
“Code Replacement Libraries”
“Code Replacement Terminology”

Introduced in R2014b

Optimization Parameters

Model Configuration Parameters: Code Generation Optimization

The **Code Generation > Optimization** category includes parameters for improving the simulation speed of your models and improving the performance of the generated code. Model configuration parameters to improve the generated code require Simulink Coder or Embedded Coder.

Parameter	Description
"Default parameter behavior" on page 15-18	Transform numeric block parameters into constant inlined values in the generated code.
"Pass reusable subsystem outputs as" on page 15-34	Specify how a reusable subsystem passes outputs.
"Remove root level I/O zero initialization" on page 15-10	Specify whether to generate initialization code for root-level inports and outports set to zero.
"Remove internal data zero initialization" on page 15-12	Specify whether to generate initialization code for internal work structures, such as block states and block outputs, to zero.
"Level" on page 15-38	Choose the optimization level that you want to apply to the generated code.
"Priority" on page 15-46	Optimize the generated code for increased execution efficiency, decreased RAM consumption, or a balance between the two.
"Specify custom optimizations" on page 15-54	Instead of applying an optimization level, select this parameter to select the optimization parameters in the **Details** section.
"Use memcpy for vector assignment" on page 15-22	Optimize code generated for vector assignment by replacing `for` loops with `memcpy`.

Parameter	Description
“Memcpy threshold (bytes)” on page 15-24	Specify the minimum array size in bytes for which `memcpy` and `memset` function calls should replace `for` loops for vector assignments in the generated code.
“Enable local block outputs” on page 15-62	Specify whether block signals are declared locally or globally.
“Reuse local block outputs” on page 15-64	Specify whether Simulink Coder software reuses signal memory.
“Eliminate superfluous local variables (Expression folding)” on page 15-66	Collapse block computations into single expressions.
“Reuse global block outputs” on page 15-68	Reuse global memory for block outputs.
“Perform inplace updates for Bus Assignment blocks” on page 15-70	Reuse the input and output variables of Bus Assignment blocks if possible.
“Reuse buffers for Data Store Read and Data Store Write blocks” on page 15-72	Remove temporary buffers for Data Store Read and Data Store Write blocks. Use the Data Store Memory block directly if possible.
“Simplify array indexing” on page 15-74	Replace multiply operations in array indices when accessing arrays in a loop.
“Pack Boolean data into bitfields” on page 15-26	Specify whether Boolean signals are stored as one–bit bitfields or as a Boolean data type.
“Bitfield declarator type specifier” on page 15-28	Specify the bitfield type when selecting configuration parameter “Pack Boolean data into bitfields” on page 15-26.
“Reuse buffers of different sizes and dimensions” on page 15-36	Reduce memory consumption by reusing buffers to store data of different sizes and dimensions.
“Optimize global data access” on page 15-78	Select global variable optimization.
“Optimize block operation order in the generated code” on page 15-76	Reorder block operations in the generated code for improved code execution speed.

Parameter	Description
“Use bitsets for storing state configuration” on page 15-56	Use bitsets to reduce the amount of memory required to store state configuration variables.
“Use bitsets for storing Boolean data” on page 15-58	Use bitsets to reduce the amount of memory required to store Boolean data.
“Maximum stack size (bytes)” on page 15-32	Specify the maximum stack size in bytes for your model.
“Loop unrolling threshold” on page 15-30	Specify the minimum signal or parameter width for which a `for` loop is generated.
“Optimize using the specified minimum and maximum values” on page 15-7	Optimize generated code using the specified minimum and maximum values for signals and parameters in the model.
Maximum number of arguments for subsystem outputs	Set maximum number of subsystem outputs to pass individually.
“Inline invariant signals” on page 15-20	Transform symbolic names of invariant signals into constant values.
“Remove code from floating-point to integer conversions with saturation that maps NaN to zero” on page 15-80	Remove code that handles floating-point to integer conversion results for `NaN` values.
“Use memset to initialize floats and doubles to 0.0” on page 15-82	Specify whether to generate code that explicitly initializes floating-point data to 0.0.
“Remove code from floating-point to integer conversions that wraps out-of-range values” on page 15-14	Remove wrapping code that handles out-of-range floating-point to integer conversion results.
“Remove code that protects against division arithmetic exceptions” on page 15-16	Specify whether to generate code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.
Buffer for reusable subsystems	Improve reuse by inserting buffers at reusable subsystem boundaries.

Parameter	Description
Disable incompatible optimizations	Specify whether to disable optimizations that are incompatible with Simulink Code Inspector.
“Base storage type for automatically created enumerations” on page 15-60	Set the storage type and size for enumerations created with active state output.
“Use signal labels to guide buffer reuse” on page 15-84	For signals with the same label, the code generator attempts to use the same signal memory.

See Also

Related Examples

- “Performance”

Optimization Pane: Tab Overview

Set up optimizations for a model's active configuration set. Optimizations are set for both simulation and code generation.

Tips

- To open the Optimization pane, in the Simulink Editor, select **Simulation > Model Configuration Parameters > Optimization**.
- Simulink Coder optimizations appear only when the Simulink Coder product is installed on your system. Selecting a GRT-based or ERT-based system target file changes the available options. ERT-based target optimizations require a Embedded Coder license when generating code. See the **Dependencies** sections below for licensing information for each parameter.

To get help on an option

1. Right-click the option text label.
2. From the context menu, select **What's This**.

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Perform Acceleration” (Simulink)
- “Performance”

Optimize using the specified minimum and maximum values

Description

Optimize generated code using the specified minimum and maximum values for signals and parameters in the model.

Category: Optimization

Settings

Default: Off

 On

Optimizes the generated code using range information derived from the minimum and maximum specified values for signals and parameters in the model.

 Off

Ignores specified minimum and maximum values when generating code.

Tips

- To detect mismatches between model and generated code simulations that arise from the use of this parameter, before running normal, accelerator, software-in-the-loop (SIL), or processor-in-the-loop (PIL) (Embedded Coder) simulations, set **Diagnostics** > **Data Validity** > **Simulation range checking** to `Warning` or `Error`.
- Specify minimum and maximum values for signals and parameters in the model for:
  - Inport and Outport blocks.
  - Block outputs.
  - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks.
  - `Simulink.Signal` objects.
- This optimization does not take into account minimum and maximum values specified for:

  - Merge block inputs. To work around this, use a `Simulink.Signal` object on the Merge block output and specify the range on this object
  - Bus elements.
  - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Outport block.

    Outport blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.

- If you use the Polyspace Code Prover™ software to verify code generated using this optimization, it might mark code that was previously green as orange. For example, if your model contains a division where the range of the denominator does not include zero, the generated code does not include protection against division by zero. Polyspace Code Prover might mark this code orange because it does not have information about the minimum and maximum values specified for the inputs to the division.

  The Polyspace Code Prover software does automatically capture some minimum and maximum values specified in the MATLAB workspace, for example, for `Simulink.Signal` and `Simulink.Parameter` objects. In this example, to provide range information to the Polyspace Code Prover software, use a `Simulink.Signal` object on the input of the division and specify a range that does not include zero.

  The Polyspace Code Prover software stores these values in a Data Range Specification (DRS) file. However, they do not capture all minimum and maximum values specified in your Simulink model. To provide additional min/max information to Polyspace Code Prover, you can manually define a DRS file. For more information, see the Polyspace Code Prover documentation.
- If you are using double-precision data types and the **Code Generation > Interface > Support non-finite numbers** configuration parameter is selected, this optimization does not occur.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different specified minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not occur. Without this optimization, the Simulink Coder software generates code once for the subsystem and shares this code among the multiple instances of the subsystem.

- The Model Advisor check Check safety-related optimization settings generates a warning if this option is selected. For many safety-critical applications, removing dead code automatically is unacceptable because doing so might make code untraceable.
- Enabling this optimization improves the ability of the Fixed-Point Designer software to eliminate unnecessary utility functions and saturation code from the generated code.

Dependencies

- This parameter appears for ERT-based targets only.
- This parameter requires a Embedded Coder license when generating code.

Command-Line Information

Parameter: `UseSpecifiedMinMax`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	Off

See Also

Related Examples

- “Optimize Generated Code Using Minimum and Maximum Values” (Embedded Coder)
- “Optimize Generated Code Using Specified Minimum and Maximum Values” (Fixed-Point Designer)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Remove root level I/O zero initialization

Description

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

Category: Optimization

Settings

Default: Off (GUI), `'on'` (command-line)

 On

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

 Off

Generates initialization code for all root-level inports and outports. Use the default:

- To initialize memory allocated for C MEX S-function wrappers to zero.
- To initialize all internal and external data to zero.

Note Generated code never initializes data of `ImportedExtern` or `ImportedExternPointer` storage classes, regardless of configuration parameter settings.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Embedded Coder license when generating code.

Command-Line Information

Parameter: `ZeroExternalMemoryAtStartup`
Value: `'off'` | `'on'`
Default: `'on'`

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	On (GUI), `off` (command line) (execution, ROM), No impact (RAM)
Safety precaution	Off (GUI), `on` (command line)

See Also

Related Examples

- “Remove Initialization Code for Root-Level Inports and Outports Set to Zero” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Performance”

Remove internal data zero initialization

Description

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

Category: Optimization

Settings

Default: Off (GUI), `'on'` (command-line)

 On

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

Selecting this parameter does not guarantee that memory is in a known state each time the generated code begins execution. When you run a model or generated S-function multiple times, each run can produce a different answer, even when calling the model initialization function in an attempt to reset memory.

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

 Off

Generates code that initializes internal work structures to zero. You should use the default:

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Embedded Coder license when generating code.

Command-Line Information

Parameter: `ZeroInternalMemoryAtStartup`
Value: `'off'` | `'on'`
Default: `'on'`

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	On (GUI), `off` (command line), (execution, ROM), No impact (RAM)
Safety precaution	Off (GUI), `on` (command line)

See Also

Related Examples

- “Remove Zero Initialization Code for Internal Data” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Performance”

Remove code from floating-point to integer conversions that wraps out-of-range values

Description

Remove wrapping code that handles out-of-range floating-point to integer conversion results.

Category: Optimization

Settings

Default: Off

On

Removes code when out-of-range conversions occur. Select this check box if code efficiency is critical to your application and the following conditions are true for at least one block in the model:

- Computing the outputs or parameters of a block involves converting floating-point data to integer or fixed-point data.
- The **Saturate on integer overflow** check box is cleared in the Block Parameters dialog box.

Caution Execution of generated code might not produce the same results as simulation.

Off

Results for simulation and execution of generated code match when out-of-range conversions occur. The generated code is larger than when you select this check box.

Tips

- Selecting this check box reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.

- Selecting this check box affects code generation results only for out-of-range values and cannot cause code generation results to differ from simulation results for in-range values.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `EfficientFloat2IntCast`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	On

See Also

Related Examples

- “Remove Code From Floating-Point to Integer Conversions That Wraps Out-of-Range Values”
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Remove code that protects against division arithmetic exceptions

Description

Specify whether to generate code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.

For more information on division arithmetic exceptions, see “Division Arithmetic Exceptions in Generated Code” (Embedded Coder).

Category: Optimization

Settings

Default: Off

 On

Does not generate code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data. To retain bit-true agreement between simulation results and results from generated code, ensure that your model never produces division by zero or `INT_MIN/-1` operations, where the quotient cannot be represented in the data type.

 Off

Generates code that guards against division by zero and `INT_MIN/-1` operations for integers and fixed-point data.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Embedded Coder license when generating code.

Command-Line Information

Parameter: `NoFixptDivByZeroProtection`
Value: `'on'` | `'off'`

Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution, ROM)
Safety precaution	Off

See Also

Related Examples

- “Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data” (Embedded Coder)
- “Division Arithmetic Exceptions in Generated Code” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Performance”

Default parameter behavior

Description

Transform numeric block parameters into constant inlined values in the generated code.

Category: Optimization

Settings

Default: `Tunable` for GRT targets | `Inlined` for ERT targets

`Inlined`

Set **Default parameter behavior** to `Inlined` to reduce global RAM usage and increase efficiency of the generated code. The code does not allocate memory to represent numeric block parameters such as the **Gain** parameter of a Gain block. Instead, the code inlines the literal numeric values of these block parameters.

`Tunable`

Set **Default parameter behavior** to `Tunable` to enable tunability of numeric block parameters in the generated code. The code represents numeric block parameters and variables that use the storage class `Auto`, including numeric MATLAB variables, as tunable fields of a global parameters structure.

Tips

- Whether you set **Default parameter behavior** to `Inlined` or to `Tunable`, create parameter data objects to preserve tunability for block parameters. For more information, see “Create Tunable Calibration Parameter in the Generated Code”.
- When you switch from a system target file that is not ERT-based to one that is ERT-based, **Default parameter behavior** sets to `Inlined` by default. However, you can change the setting of **Default parameter behavior** later.
- When a top model uses referenced models or if a model is referenced by another model:
  - All referenced models must set **Default parameter behavior** to `Inlined` if the top model has **Default parameter behavior** set to `Inlined`.
  - The top model can specify **Default parameter behavior** as `Tunable` or `Inlined`.

- If your model contains an Environment Controller block, you can suppress code generation for the branch connected to the Sim port if you set **Default parameter behavior** to `Inlined` and the branch does not contain external signals.

Dependencies

When you set **Default parameter behavior** to `Inlined`, you enable "Inline invariant signals" on page 15-20 configuration parameter.

Command-Line Information

Parameter: `DefaultParameterBehavior`
Type: character vector
Value: `'Inlined'` | `'Tunable'`
Default: `'Tunable'` for GRT targets | `'Inlined'` for ERT targets

Recommended Settings

Application	Setting
Debugging	`Tunable` during development `Inlined` for production code generation
Traceability	`Inlined`
Efficiency	`Inlined`
Safety precaution	No impact

See Also

Related Examples

- "Model Configuration Parameters: Code Generation Optimization" on page 15-2
- "Inline Numeric Values of Block Parameters"
- "How Generated Code Stores Internal Signal, State, and Parameter Data"

Inline invariant signals

Description

Transform symbolic names of invariant signals into constant values.

Category: Optimization

Settings

Default: Off

On

Simulink Coder software uses the numerical values of model parameters, instead of their symbolic names, in generated code. An invariant signal is not inline if it is nonscalar, complex, or the block inport the signal is attached to takes the address of the signal.

Off

Uses symbolic names of model parameters in generated code.

Dependencies

- This parameter requires a Simulink Coder license.
- This parameter is enabled when you set **Default parameter behavior** to `Inlined`.

Command-Line Information

Parameter: `InlineInvariantSignals`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off

Application	Setting
Traceability	Off
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Inline Invariant Signals”
- “Performance”

Use memcpy for vector assignment

Description

Optimize code generated for vector assignment by replacing `for` loops with `memcpy`.

Category: Optimization

Settings

Default: On

On

Enables use of `memcpy` for vector assignment based on the associated threshold parameter **Memcpy threshold (bytes)**. `memcpy` is used in the generated code if the number of array elements times the number of bytes per element is greater than or equal to the specified value for **Memcpy threshold (bytes)**. One byte equals the width of a character in this context.

Off

Disables use of `memcpy` for vector assignment.

Dependencies

- This parameter requires a Simulink Coder license.
- When selected, this parameter enables the associated parameter **Memcpy threshold (bytes)**.

Command-Line Information

Parameter: `EnableMemcpy`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Use memcpy Function to Optimize Generated Code for Vector Assignments”
- “Performance”

Memcpy threshold (bytes)

Description

Specify the minimum array size in bytes for which `memcpy` and `memset` function calls should replace `for` loops for vector assignments in the generated code.

Category: Optimization

Settings

Default: `64`

Dependencies

- This parameter requires a Simulink Coder license.
- For the `memcpy` optimization, this parameter is enabled when you select **Use memcpy for vector assignment**.

Command-Line Information

Parameter: `MemcpyThreshold`
Type: integer
Value: any valid quantity of bytes
Default: `64`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Accept default or determine target-specific optimal value
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Use memcpy Function to Optimize Generated Code for Vector Assignments”
- “Performance”

Pack Boolean data into bitfields

Description

Specify whether Boolean signals are stored as one–bit bitfields or as a Boolean data type.

Category: Optimization

Note You cannot use this optimization when you generate code for a target that specifies an explicit structure alignment.

Settings

Default: Off

 On

Stores Boolean signals into one–bit bitfields in global block I/O structures or DWork vectors. This will reduce RAM, but might cause more executable code.

 Off

Stores Boolean signals as a Boolean data type in global block I/O structures or DWork vectors.

Dependencies

This parameter:

- Requires a Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `BooleansAsBitfields`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), On (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Optimize Generated Code By Packing Boolean Data Into Bitfields” (Embedded Coder)
- “Bitfield declarator type specifier” on page 15-28
- “Performance” (Embedded Coder)

Bitfield declarator type specifier

Description

Specify the bitfield type when selecting configuration parameter “Pack Boolean data into bitfields” on page 15-26.

Category: Optimization

Note The optimization benefit is dependent upon your choice of target.

Settings

Default: `uint_T`

uint_T

The type specified for a bitfield declaration is an `unsigned int`.

uchar_T

The type specified for a bitfield declaration is an `unsigned char`.

Tip

The “Pack Boolean data into bitfields” on page 15-26 configuration parameter default setting uses unsigned integers. This might cause an increase in RAM if the bitfields are small and distributed. In this case, `uchar_T` might use less RAM depending on your target.

Dependency

Pack Boolean data into bitfields enables this parameter.

Command-Line Information

Parameter: `BitfieldContainerType`
Value: `uint_T | uchar_T`

Default: `uint_T`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Target dependent
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Performance” (Embedded Coder)

Loop unrolling threshold

Description

Specify the minimum signal or parameter width for which a `for` loop is generated.

Category: Optimization

Settings

Default: 5

Specify the array size at which the code generator begins to use a `for` loop instead of separate assignment statements to assign values to the elements of a signal or parameter array.

When there are perfectly nested loops, the code generator uses a for loop if the product of the loop counts for all loops in the perfect loop nest is greater than or equal to the threshold.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `RollThreshold`
Type: character vector
Value: any valid value
Default: `'5'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	>0
Safety precaution	>1

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Configure Loop Unrolling Threshold”
- “Performance”

Maximum stack size (bytes)

Description

Specify the maximum stack size in bytes for your model.

Category: Optimization

Settings

Default:`Inherit from target`

`Inherit from target`

The Simulink Coder software assigns the maximum stack size to the smaller value of the following:

- The default value (200,000 bytes) set by the Simulink Coder software
- Value of the TLC variable `MaxStackSize` in the system target file

`<Specify a value>`

Specify a positive integer value. Simulink Coder software assigns the maximum stack size to the specified value.

Note If you specify a maximum stack size for a model, the estimated required stack size of a referenced model must be less than the specified maximum stack size of the parent model.

Tips

- If you specify the maximum stack size to be zero, then the generated code implements all variables as global data.
- If you specify the maximum stack to be `inf`, then the generated code contains the least number of global variables.
- If your model contains a variable that is larger than 4096 bytes, the code generator implements it in global memory by default. You can increase the size of variables that the code generator places in local memory by changing the value of the TLC variable

`MaxStackVariableSize`. You can change this value by typing the following command in MATLAB Command Window:
`set_param(modelName,'TLCOptions','-aMaxStackVariableSize=N')`

Command-Line Information

Parameter: `MaxStackSize`
Type: `int`
Value: Any valid value
Default: `Inherit from target`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Customize Stack Space Allocation”
- “Performance”

Pass reusable subsystem outputs as

Description

Specify how a reusable subsystem passes outputs.

Category: Optimization

Settings

Default: `Individual arguments`

`Individual arguments`

Passes each reusable subsystem output argument as an address of a local, instead of as a pointer to an area of global memory containing all output arguments. This option reduces global memory usage and eliminates copying local variables back to global block I/O structures. When the signals are allocated as local variables, there may be an increase in stack size. If the stack size increases beyond a level that you want, use the default setting. By default, the maximum number of output arguments passed individually is 12. To increase the number of arguments, increase the value of the **Maximum number of arguments for subsystem outputs** parameter.

`Structure reference`

Passes reusable subsystem outputs as a pointer to a structure stored in global memory.

Note The default option is used for reusable subsystems that have signals with variable dimensions.

Dependencies

This parameter:

- Requires a Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `PassReuseOutputArgsAs`
Value: `'Structure reference'` | `'Individual arguments'`
Default: `'Individual arguments'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`Individual arguments` (execution, RAM),`Structure reference` (ROM)
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Generate Reusable Code from Library Subsystems Shared Across Models”
- “Optimize Generated Code By Passing Reusable Subsystem Outputs as Individual Arguments” (Embedded Coder)
- “Performance” (Embedded Coder)

Reuse buffers of different sizes and dimensions

Reduce memory consumption by reusing buffers to store data of different sizes and dimensions.

Settings

Default: On

 On

The code generator tries to reuse the same buffers to store data of different sizes and dimensions. This optimization conserves RAM and ROM consumption.

 Off

The code generator reuses buffers only if they have the same size and shape as the data.

Dependencies

- This parameter appears only for ERT-based targets.
- When generating code, this parameter requires an Embedded Coder license.
- This parameter is enabled by “Signal storage reuse” (Simulink).

Tips

- If your model contains a reusable custom storage class to specify reuse on signals that have different sizes and shapes, you must select the **Reuse buffers of different sizes and dimensions** parameter or remove the specification. Otherwise, during simulation, the model produces an error.
- The code generator does not replace a buffer with a lower priority buffer that has a smaller size.
- The code generator does not reuse buffers that have different sizes and symbolic dimensions.

Command-Line Information

Parameter: `DifferentSizesBufferReuse`

Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	`off`
Traceability	`off`
Efficiency	`on`
Safety precaution	No impact

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Reuse Buffers of Different Sizes and Dimensions” (Embedded Coder)

Level

Description

Choose the optimization level that you want to apply to the generated code.

Settings

Default: `Maximum`

`Minimum (Debugging)`

Configure code generation settings for debugging.

`Balanced with Readability`

Apply code generation optimizations that balance RAM efficiency and execution speed with the readability of the generated code. For example, selecting this value disables optimizations that cross atomic subsystem boundaries.

`Maximum`

Configure code generation settings based on your code efficiency objectives. Choosing this setting enables the **Priority** parameter. Set the **Priority** parameter to one of these values:

- `Balance RAM and speed (default setting)`
- `Maximum execution speed`
- `Minimize RAM`

Dependencies

- This parameter appears only for ERT-based targets.
- When generating code, this parameter requires an Embedded Coder license.

Tips

For each **Priority** and **Level** parameter value, there are corresponding values for the parameters in the **Details** section. These are some important differences among these various settings:

- If you set the **Level** parameter to `Minimum (debugging)`, all parameters in the **Details** section are set to off. The code generator does not implement optimizations that remove variables or code making it easier to debug the generated code.
- The parameter settings for `Balanced with Readability` and `Balance RAM and speed` are the same except for these three parameters:

  - **Reuse buffers of different sizes and dimensions**
  - **Optimize global data access**
  - **Optimize block operation order in the generated code**

  The above optimizations can potentially hurt readability because they cross atomic subsystem boundaries and **Optimize block operation order in the generated code** might change the block execution order in the generated code so that it is different than in simulation.
- If you have limited RAM, choose the `Minimize RAM` setting. This setting enables these optimizations that reduce RAM at the expense of a potential slow-down in execution speed:

  - **Pack Boolean data into bitfields**
  - **Reuse buffers of different sizes and dimensions**
  - **Use bitsets for storing state configuration**
  - **Use bitsets for storing Boolean data**

  This setting also changes the **Optimize block operation order in the generated code** from `Improved Code Execution Speed` to `off`.

For each **Priority** and **Level** parameter value, this table lists the corresponding values for the parameters in the **Details** section.

Parameters	Settings			Example
Level	`Minimum (debugging)`	`Balanced with readability`	`Maximum`	

Parameters	Settings					Example
Priority	Not Applicable (N/A)	N/A	`Balance RAM and speed`	`Maximize execution speed`	`Minimize RAM`	
Details						
Use memcpy for vector assignment	`Off`	`On`	`On`	`On`	`On`	"Use memcpy Function to Optimize Generated Code for Vector Assignments"
Memcpy threshold (bytes)	`Off`	`64`	`64`	`64`	`64`	"Use memcpy Function to Optimize Generated Code for Vector Assignments"
Enable local block outputs	`Off`	`On`	`On`	`On`	`On`	"Enable and Reuse Local Block Outputs in Generated Code"

Parameters	Settings					Example
Reuse local block outputs	`Off`	`On`	`On`	`On`	`On`	“Enable and Reuse Local Block Outputs in Generated Code”
Eliminate superfluous local variables (expression folding)	`Off`	`On`	`On`	`On`	`On`	“Minimize Computations and Storage for Intermediate Results at Block Outputs”
Reuse global block outputs	`Off`	`On`	`On`	`On`	`On`	“Reuse Global Block Outputs in the Generated Code” (Embedded Coder)
Perform inplace updates for Bus Assignment blocks	`Off`	`On`	`on`	`On`	`On`	“Data copy reduction for Bus Assignment block” (Embedded Coder)

Parameters	Settings					Example
Reuse buffers for Data Store Read and Data Store Write blocks	`Off`	`On`	`On`	`On`	`On`	“Data Copy Reduction for Data Store Read and Data Store Write Blocks” (Embedded Coder)
Simplify array indexing	`Off`	`Off`	`Off`	`On`	`Off`	“Simplify Multiply Operations in Array Indexing” (Embedded Coder)
Pack Boolean data into bitfields	`Off`	`Off`	`Off`	`Off`	`On`	“Optimize Generated Code By Packing Boolean Data Into Bitfields” (Embedded Coder)
Reuse buffers of different sizes and dimensions	`Off`	`Off`	`On`	`Off`	`On`	“Reuse Buffers of Different Sizes and Dimensions” (Embedded Coder)

Parameters	Settings					Example
Optimize global data access	None	None	Use global to hold temporary results	None	Use global to hold temporary results	“Optimize Global Variable Usage” (Embedded Coder)
Optimize block operation order in the generated code	Off	Off	Improved Code Execution Speed	Improved Code Execution Speed	Off	“Remove Data Copies by Reordering Block Operations in the Generated Code” (Embedded Coder)
Use bitsets for storing state configuration	Off	Off	Off	Off	On	“Reduce Memory Usage for Boolean and State Configuration Variables” (Embedded Coder)

Parameters	Settings					Example
Use bitsets for storing Boolean data	`Off`	`Off`	`Off`	`Off`	`On`	"Reduce Memory Usage for Boolean and State Configuration Variables" (Embedded Coder)

If you plan on upgrading your software, be aware that:

- Setting the **Level** and **Priority** parameters enables the latest optimizations corresponding with the above parameter settings for each subsequent release.
- Selecting the **Specify custom optimizations** parameter enables you to select individual parameters in the **Details** section. When you load a model in a future release, any optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to `off`. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Command-Line Information

Parameter: `OptimizationPriority`
Value: `'Minimum (Debugging)'|'Balanced with Readability'|'Maximum'`
Default: `'Maximum'`

Recommended Settings

Application	Setting
Debugging	`Minimum (debugging)`
Traceability	`Minimum (debugging)`
Efficiency	Based on your goals, choose `Balanced with Readability` or `Maximum`. If you choose `Maximum`, set the **Priority** parameter.

Application	Setting
Safety precaution	No impact

See Also

“Priority” on page 15-46 | “Specify custom optimizations” on page 15-54

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Priority

Description

Optimize the generated code for increased execution efficiency, decreased RAM consumption, or a balance between execution efficiency and RAM consumption.

Settings

Default: `Balance RAM and speed`

`Balance RAM and speed`

Configure code generation settings to balance RAM and execution speed.

`Maximize execution speed`

Apply code generation settings to maximize execution speed.

`Minimize RAM`

Configure code generation settings to minimize RAM consumption.

Dependencies

- Enable this parameter by setting the **Level** parameter to `Maximum`.
- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Tips

For each **Priority** and **Level** parameter value, there are corresponding values for the parameters in the **Details** section. These are some important differences among these various settings:

- If you set the **Level** parameter to `Minimum (debugging)`, all parameters in the **Details** section are set to off. The code generator does not implement optimizations that remove variables or code making it easier to debug the generated code.
- The parameter settings for `Balanced with Readability` and `Balance RAM and speed` are the same except for these three parameters:

- **Reuse buffers of different sizes and dimensions**
- **Optimize global data access**
- **Optimize block operation order in the generated code**

The above optimizations can potentially hurt readability because they cross atomic subsystem boundaries and **Optimize block operation order in the generated code** might change the block execution order in the generated code so that it is different than in simulation.

- If you have limited RAM, choose the `Minimize RAM` setting. This setting enables these optimizations that reduce RAM at the expense of a potential slow-down in execution speed:

  - **Pack Boolean data into bitfields**
  - **Reuse buffers of different sizes and dimensions**
  - **Use bitsets for storing state configuration**
  - **Use bitsets for storing Boolean data**

  This setting also changes the **Optimize block operation order in the generated code** from `Improved Code Execution Speed` to `off`.

For each **Priority** and **Level** parameter value, this table lists the corresponding values for the parameters in the **Details** section.

Parameters	**Settings**					**Example**
Level	`Minimum (debugging)`	`Balanced with readability`	`Maximum`			
Priority	Not Applicable (N/A)	N/A	`Balance RAM and speed`	`Maximize execution speed`	`Minimize RAM`	
Details						

Parameters	Settings					Example
Use memcpy for vector assignment	`Off`	`On`	`On`	`On`	`On`	“Use memcpy Function to Optimize Generated Code for Vector Assignments”
Memcpy threshold (bytes)	`Off`	`64`	`64`	`64`	`64`	“Use memcpy Function to Optimize Generated Code for Vector Assignments”
Enable local block outputs	`Off`	`On`	`On`	`On`	`On`	“Enable and Reuse Local Block Outputs in Generated Code”
Reuse local block outputs	`Off`	`On`	`On`	`On`	`On`	“Enable and Reuse Local Block Outputs in Generated Code”

Parameters	Settings					Example
Eliminate superfluous local variables (expression folding)	`Off`	`On`	`On`	`On`	`On`	“Minimize Computations and Storage for Intermediate Results at Block Outputs”
Reuse global block outputs	`Off`	`On`	`On`	`On`	`On`	“Reuse Global Block Outputs in the Generated Code” (Embedded Coder)
Perform inplace updates for Bus Assignment blocks	`Off`	`On`	`on`	`On`	`On`	“Data copy reduction for Bus Assignment block” (Embedded Coder)

Parameters	Settings					Example
Reuse buffers for Data Store Read and Data Store Write blocks	`Off`	`On`	`On`	`On`	`On`	“Data Copy Reduction for Data Store Read and Data Store Write Blocks” (Embedded Coder)
Simplify array indexing	`Off`	`Off`	`Off`	`On`	`Off`	“Simplify Multiply Operations in Array Indexing” (Embedded Coder)
Pack Boolean data into bitfields	`Off`	`Off`	`Off`	`Off`	`On`	“Optimize Generated Code By Packing Boolean Data Into Bitfields” (Embedded Coder)
Reuse buffers of different sizes and dimensions	`Off`	`Off`	`On`	`Off`	`On`	“Reuse Buffers of Different Sizes and Dimensions” (Embedded Coder)

Parameters	Settings					Example
Optimize global data access	None	None	Use global to hold temporary results	None	Use global to hold temporary results	“Optimize Global Variable Usage” (Embedded Coder)
Optimize block operation order in the generated code	Off	Off	Improved Code Execution Speed	Improved Code Execution Speed	Off	“Remove Data Copies by Reordering Block Operations in the Generated Code” (Embedded Coder)
Use bitsets for storing state configuration	Off	Off	Off	Off	On	“Reduce Memory Usage for Boolean and State Configuration Variables” (Embedded Coder)

Parameters	Settings					Example
Use bitsets for storing Boolean data	`Off`	`Off`	`Off`	`Off`	`On`	"Reduce Memory Usage for Boolean and State Configuration Variables" (Embedded Coder)

If you plan on upgrading your software, be aware that:

- Setting the **Level** and **Priority** parameters enables the latest optimizations corresponding with the above parameter settings for each subsequent release.
- Selecting the **Specify custom optimizations** parameter enables you to select individual parameters in the **Details** section. When you load a model in a future release, any optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to `off`. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Command-Line Information

Parameter: `OptimizationPriority`
Value: `'Balance RAM and speed'` | `'Maximize execution speed'` | `'Minimize RAM'`
Default: `'Balance RAM and speed'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	`Minimize RAM` (RAM), `Maximum execution speed` (execution speed), `Balance RAM and speed` (RAM and execution speed)
Safety precaution	No impact

See Also

“Level” on page 15-38 | “Specify custom optimizations” on page 15-54

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Specify custom optimizations

Description

Select this parameter to enable the optimization parameters in the **Details** section. Selecting this parameter enables you to choose individual optimization parameters rather than setting the **Priority** and **Level** parameters.

Settings

Default: Off

 On

Enables you to individually select or clear parameters in the **Details** section.

 Off

Disables the parameters in the **Details** section, so that you cannot individually select or clear these parameters.

Clearing this parameter enables the **Level** and **Priority** parameters and resets all parameters in the **Details** section to the current settings for the **Level** and **Priority** parameters.

Dependencies

This parameter:

- Enables parameters in the **Details** section.
- Requires an Embedded Coder license.
- Appears only for ERT-based targets.

Tips

If you plan on upgrading your software, be aware that:

- Setting the **Priority** and **Level** parameters enables the latest optimizations corresponding with these settings for each subsequent release.

- Selecting **Specify custom optimizations** means that when you load a model in a future release, any optimization parameters that were introduced in releases after you adopted the software to when you upgrade are set to `off`. If you want to reduce the number of changes in the generated code when you upgrade your software, this option can be a good choice.

Command-Line Information

Parameter: `OptimizationCustomize`
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	`Off` and set **Level** parameter to `Minimize (debugging)`
Traceability	`Off` and set **Level** parameter to `Minimize (debugging)`
Efficiency	Depends on individual parameter settings
Safety precaution	No impact

See Also

“Level” on page 15-38 | “Priority” on page 15-46

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Use bitsets for storing state configuration

Description

Use bitsets to reduce the amount of memory required to store state configuration variables.

Category: Optimization

Settings

Default: Off

 On

Stores state configuration variables in bitsets. Potentially reduces the amount of memory required to store the variables. Potentially requires more instructions to access state configuration, which can result in less optimal code.

 Off

Stores state configuration variables in unsigned bytes. Potentially increases the amount of memory required to store the variables. Potentially requires fewer instructions to access state configuration, which can result in more optimal code.

Tips

- Selecting this check box can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.
- Select this check box for Stateflow charts that have a large number of sibling states at a given level of the hierarchy.
- Clear this check box for Stateflow charts with a small number of sibling states at a given level of the hierarchy.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `StateBitsets`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	Off (execution, ROM), On (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Reduce Memory Usage for Boolean and State Configuration Variables”
- “Design Tips for Optimizing Generated Code for Stateflow Objects”

Use bitsets for storing Boolean data

Description

Use bitsets to reduce the amount of memory required to store Boolean data.

Category: Optimization

Settings

Default: Off

 On

Stores Boolean data in bitsets. Potentially reduces the amount of memory required to store the data. Potentially requires more instructions to access the data, which can result in less optimal code.

 Off

Stores Boolean data in unsigned bytes. Potentially increases the amount of memory required to store the data. Potentially requires fewer instructions to access the data, which can result in more optimal code.

Tips

- Select this check box for Stateflow charts that reference Boolean data infrequently.
- Clear this check box for Stateflow charts that reference Boolean data frequently.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `DataBitsets`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	Off (execution, ROM), On (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Reduce Memory Usage for Boolean and State Configuration Variables”
- “Design Tips for Optimizing Generated Code for Stateflow Objects”

Base storage type for automatically created enumerations

Description

Set the storage type and size for enumerations created with active state output.

Category: Optimization

Settings

Default: `'Native Integer'`

`'Native Integer'`
: Default target integer type

`int32`
: 32 bit signed integer type

`int16`
: 16 bit signed integer type

`int8`
: 8 bit signed integer type

`uint16`
: 16 bit unsigned integer type

`uint8`
: 8 bit unsigned integer type

Tips

- The default `'Native Integer'` is recommended for most models.
- If you need a smaller memory footprint for the generated enumerations, set the storage type to a smaller size. The size must be large enough to hold the number of states in the chart.

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `ActiveStateOutputEnumStorageType`
Value: `'Native Integer'|'int32'|'int16'|'int8'|'uint16'|'uint8'`
Default: `'Native Integer'`

See Also

Related Examples

- “Monitor State Activity Through Active State Data” (Stateflow)
- “Design Tips for Optimizing Generated Code for Stateflow Objects”

Enable local block outputs

Description

Specify whether block signals are declared locally or globally.

Category: Optimization

Settings

Default: On

On

Block signals are declared locally in functions.

Off

Block signals are declared globally.

Tips

- If it is not possible to declare an output as a local variable, the generated code declares the output as a global variable.
- If you are constrained by limited stack space, you can turn **Enable local block outputs** off and still benefit from memory reuse.

Dependencies

- This parameter requires a Simulink Coder license.
- This parameter is enabled by **Signal storage reuse**.

Command-Line Information

Parameter: `LocalBlockOutputs`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Enable and Reuse Local Block Outputs in Generated Code”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Reuse local block outputs

Description

Specify whether Simulink Coder software reuses signal memory.

Category: Optimization

Settings

Default: On

On

- Simulink Coder software reuses signal memory whenever possible, reducing stack size where signals are being buffered in local variables.
- Selecting this parameter trades code traceability for code efficiency.

Off

Signals are stored in unique locations.

Dependencies

This parameter:

- Is enabled by **Signal storage reuse**.
- Requires a Simulink Coder license.

Command-Line Information

Parameter: `BufferReuse`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Enable and Reuse Local Block Outputs in Generated Code”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Eliminate superfluous local variables (Expression folding)

Description

Collapse block computations into single expressions.

Category: Optimization

Settings

Default: On

On

- Enables expression folding.
- Eliminates local variables, incorporating the information into the main code statement.
- Improves code readability and efficiency.

Off

Disables expression folding.

Dependencies

- This parameter requires a Simulink Coder license.
- This parameter is enabled by **Signal storage reuse**.

Command-Line Information

Parameter: `ExpressionFolding`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact for simulation or during development Off for production code generation
Efficiency	On
Safety precaution	No impact

See Also

Related Examples

- “Minimize Computations and Storage for Intermediate Results at Block Outputs”
- “Performance”
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Reuse global block outputs

Description

Reuse global memory for block outputs.

Category: Optimization

Settings

Default: On

On

- Software reuses signal memory whenever possible, reducing global variable use.
- Selecting this parameter trades code traceability for code efficiency.

Off

Signals are stored in unique locations.

Dependencies

This parameter:

- Is enabled by “Signal storage reuse” (Simulink) .
- Requires an Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `GlobalBufferReuse`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM, RAM)
Safety precaution	No impact

See Also

Related Examples

- “Reuse Global Block Outputs in the Generated Code” (Embedded Coder)
- “Performance” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Perform inplace updates for Bus Assignment blocks

Description

Reuse the input and output variables of Bus Assignment and Assignment blocks if possible.

Category: Optimization

Settings

Default: On

 On

Embedded Coder reuses the input and output variables of Bus Assignment and Assignment blocks if possible. Reusing these variables reduces data copies, conserves RAM consumption and increases code execution speed.

 Off

Embedded Coder does not reuse the input and output variables of Bus Assignment and Assignment blocks.

Dependency

- The parameter **Signal Storage Reuse** enables this parameter.
- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Command-Line Information

Parameter: `BusAssignmentInplaceUpdate`
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Model Configuration Parameters: Code Generation Optimization” on page 15-2
- “Reduce Data Copies for Bus Assignment Blocks” (Embedded Coder)

Reuse buffers for Data Store Read and Data Store Write blocks

Description

Remove temporary buffers for Data Store Read and Data Store Write blocks. Use the Data Store Memory block directly if possible.

Category: Optimization

Settings

Default: On

 On

Embedded Coder reads directly from the Data Store Memory block and writes directly to the Data Store Memory block, if possible. Using the Data Store Memory block directly eliminates data copies in the generated code, conserving RAM consumption and increasing code execution speed.

 Off

Embedded Coder inserts buffers in the generated code for Data Store Read and Data Store Write blocks.

Dependency

- The parameter **Signal Storage Reuse** enables this parameter.
- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Command-Line Information

Parameter: `OptimizeDataStoreBuffers`
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Data Copy Reduction for Data Store Read and Data Store Write Blocks” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Simplify array indexing

Description

Replace multiply operations in array indices when accessing arrays in a loop.

Category: Optimization

Settings

Default: Off

 On

In array indices, replace multiply operations with add operations when accessing arrays in a loop in the generated code. When the original signal is multidimensional, the Embedded Coder generates one-dimensional arrays, resulting in multiply operations in the array indices. Using this setting eliminates costly multiply operations when accessing arrays in a loop in the C/C++ program. This optimization (commonly referred to as strength reduction) is particularly useful if the C/C++ compiler on the target platform does not have similar functionality. No appearance of multiply operations in the C/C++ program does not imply that the C/C++ compiler does not generate multiply instructions.

 Off

Leave multiply operations in array indices when accessing arrays in a loop.

Dependencies

This parameter:

- Requires a Embedded Coder license to generate code.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `StrengthReduction`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (execution speed)
Safety precaution	No impact

See Also

Related Examples

- “Simplify Multiply Operations in Array Indexing” (Embedded Coder)
- “Performance” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Optimize block operation order in the generated code

Description

Reorder block operations in the generated code for improved code execution speed.

Category: Optimization

Settings

Default: Improved Code Execution Speed

Off

Embedded Coder does not reorder block operation order in the generated code to create additional instances of buffer reuse.

Improved Code Execution Speed

Embedded Coder changes the block operation order in the generated code so that more instances of buffer reuse can occur. Reusing buffers conserves RAM and ROM consumption and improves code execution speed.

Dependency

- The parameter **Signal Storage Reuse** enables this parameter.
- This parameter requires an Embedded Coder license.
- This parameter appears only for ERT-based targets.

Command-Line Information

Parameter: `OptimizeBlockOrder`
Value: `'Speed' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Improved Code Execution Speed
Safety precaution	No recommendation

See Also

Related Examples

- “Remove Data Copies by Reordering Block Operations in the Generated Code” (Embedded Coder)
- “Improve Execution Efficiency by Reordering Block Operations in the Generated Code” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Optimize global data access

Description

Select global variable optimization.

Category: Optimization

Settings

Default: `Use global to hold temporary results`

`None`

Use default optimizations.

`Use global to hold temporary results`

Maximize use of global variables.

`Minimize global data access`

Minimize use of global variables by using local variables to hold intermediate values.

Dependencies

- This parameter is enabled by **“Signal storage reuse” (Simulink)**.
- This parameter requires an Embedded Coder license.
- Appears only for ERT-based targets.

Command-Line Information

Parameter: `GlobalVariableUsage`
Value: `'None'|'Use global to hold temporary results'|'Minimize global data access'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	`'Use global to hold temporary results'` (RAM), `'Minimize global data access'` (ROM)
Safety precaution	No impact

See Also

Related Examples

- “Optimize Global Variable Usage” (Embedded Coder)
- “Performance” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Remove code from floating-point to integer conversions with saturation that maps NaN to zero

Description

Remove code that handles floating-point to integer conversion results for `NaN` values.

Category: Optimization

Settings

Default: On

On

Removes code when mapping from `NaN` to integer zero occurs. Select this check box if code efficiency is critical to your application and the following conditions are true for at least one block in the model:

- Computing outputs or parameters of a block involves converting floating-point data to integer or fixed-point data.
- The **Saturate on integer overflow** check box is selected in the Block Parameters dialog box.

Caution Execution of generated code might not produce the same results as simulation.

Off

Results for simulation and execution of generated code match when mapping from `NaN` to integer zero occurs. The generated code is larger than when you select this check box.

Tips

- Selecting this check box reduces the size and increases the speed of the generated code at the cost of producing results that do not match simulation in the case of `NaN` values.

- Selecting this check box affects code generation results only for `NaN` values and cannot cause code generation results to differ from simulation results for any other values.

Dependencies

- This parameter requires a Simulink Coder license.
- For ERT-based targets, this parameter is enabled when you select the **floating-point numbers** and **non-finite numbers** check boxes in the **Code Generation** > **Interface** pane.

Command-Line Information

Parameter: `EfficientMapNaN2IntZero`
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On
Safety precaution	No recommendation

See Also

Related Examples

- “Remove Code That Maps NaN to Integer Zero”
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Use memset to initialize floats and doubles to 0.0

Description

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

Category: Optimization

Settings

Default: On (GUI), `'off'` (command-line)

 On

Uses `memset` to clear internal storage for floating-point data to integer bit pattern 0 (all bits 0), regardless of type. If your compiler and target CPU both represent floating-point zero with the integer bit pattern `0`, use this parameter to gain execution and ROM efficiency.

 Off

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

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

Dependency

This parameter requires a Simulink Coder license.

Command-Line Information

Parameter: `InitFltsAndDblsToZero`
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	On (GUI), `'off'` (command-line) (execution, ROM), No impact (RAM)
Safety precaution	No impact

See Also

Related Examples

- “Optimize Generated Code Using memset Function”
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2

Use signal labels to guide buffer reuse

Description

For signals with the same label, the code generator attempts to use the same signal memory.

Category: Optimization

Settings

Default: Off

 On

The code generator uses signal labels as a guide for which buffers to reuse.

 Off

The code generator ignores signal labels when implementing buffer reuse.

Dependencies

This parameter:

- Is enabled by “Signal storage reuse” (Simulink) .
- Requires an Embedded Coder license.
- Appears only for ERT-based targets.

Tips

If your model has the optimal parameter settings for removing data copies, you might be able to remove additional data copies by using signal labels. After studying the generated code and the Static Code Metrics Report and identifying blocks for whoose input and output signals you would like to reuse, you can add labels to signal lines and enable the **Use signal labels to guide buffer reuse** parameter. If possible, the code generator implements the reuse specification.

Command-Line Information

Parameter:`LabelGuidedReuse`
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	Off
Efficiency	On (execution, ROM, RAM)
Safety precaution	No impact

See Also

Related Examples

- “Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse” (Embedded Coder)
- “Model Configuration Parameters: Code Generation Optimization” on page 15-2