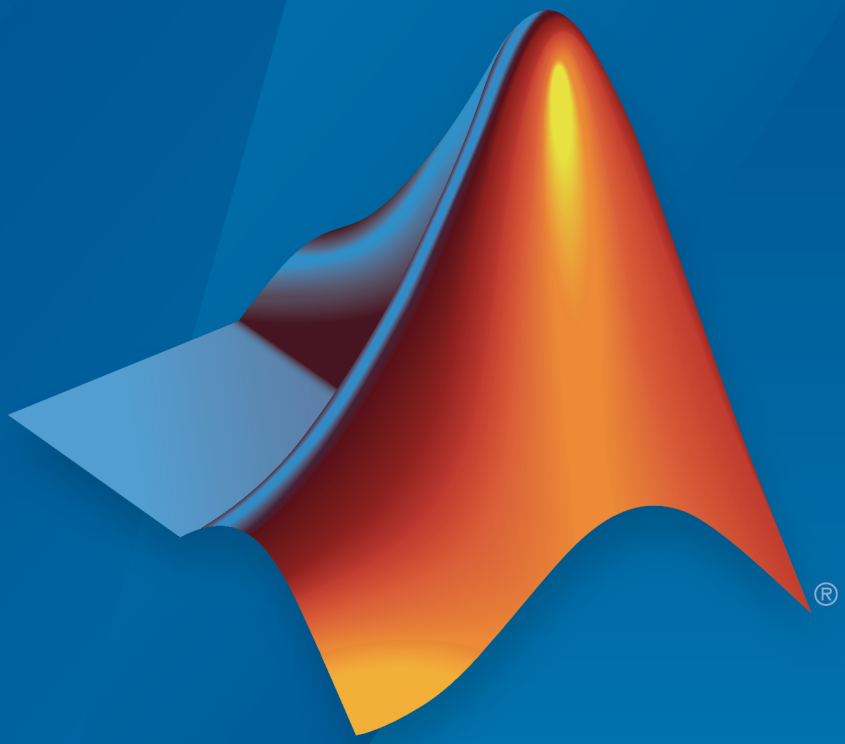
Simulink® Coder™

Reference



MATLAB® & SIMULINK®

R2016b

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)

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

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: All Parameters Tab Only

11 Configuration Parameters for Simulink Models

12 Model Advisor Checks

13 Parameters for Creating Protected Models

14 Tools — Alphabetical List

1

Simulink Code Generation Limitations

Simulink Code Generation Limitations

The following topics identify Simulink code generation limitations:

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

Alphabetical List

addCompileFlags

Add compiler options to model build information

Syntax

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

groups is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

options

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

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

groups (optional)

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

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

You can apply

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

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

Note:

- To specify compiler options to be used in the standard code generator makefile build process, specify *groups* as either `'OPTS'` or `'OPT_OPTS'`.

- To control compiler optimizations for the code generator makefile build at Simulink GUI level, use the **Compiler optimization level** parameter on the **All Parameters** tab of the Simulink Configuration Parameters dialog box. The **Compiler optimization level** parameter provides
  - Target-independent values `Optimizations on (faster runs)` and `Optimizations off (faster builds)`, which allow you to easily toggle compiler optimizations on and off during code development
  - The value `Custom` for entering custom compiler optimization flags at Simulink GUI level (rather than at other levels of the build process)

  If you use the configuration parameter **Make command** to specify compiler options for the code generator 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.

Description

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

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

Examples

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addDefines` | `addLinkFlags` | `getCompileFlags`

Introduced in R2006a

addDefines

Add preprocessor macro definitions to model build information

Syntax

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

groups is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

macrodefs

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

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

groups (optional)

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

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

You can apply

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

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

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

Description

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

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

Examples

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addCompileFlags` | `addLinkFlags` | `getDefines`

Introduced in R2006a

addIncludeFiles

Add include files to model build information

Syntax

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

paths and *groups* are optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

filenames

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

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

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

The function removes duplicate include file entries that

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

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

paths (optional)

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

groups (optional)

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

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

You can apply

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

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

Description

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

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

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

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

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

Examples

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

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

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

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

- Add the include files `etc.h`, `etc_private.h`, and `mytypes.h` to build information `myModelBuildInfo`. Group the files `etc.h` and `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 the `.h` files in a specified folder to build information `myModelBuildInfo` and place the files in the group `HFiles`.

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

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

Introduced in R2006a

addIncludePaths

Add include paths to model build information

Syntax

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

`groups` is optional.

Arguments

`buildinfo`

Build information returned by `RTW.BuildInfo`.

`paths`

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

The function removes duplicate include file entries that

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

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

`groups` (optional)

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

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

You can apply

- A single group name to an include path

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

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

Description

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

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

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

Examples

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

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo,...
```

```
'/etcproj/etc/etc_build');
```

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

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

- Add the include paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to build information `myModelBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/etc_build` with the 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'});
```

More About

- "Customize Post-Code-Generation Build Processing"

See Also

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

Introduced in R2006a

addLinkFlags

Add link options to model build information

Syntax

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

groups is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

options

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

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

groups (optional)

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

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

You can apply

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

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

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

Description

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

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

Examples

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

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

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

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

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

  ```
  myModelBuildInfo = RTW.BuildInfo;
  ```

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addCompileFlags` | `addDefines` | `getLinkFlags`

Introduced in R2006a

addLinkObjects

Add link objects to model build information

Syntax

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

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

Arguments

`buildinfo`

Build information returned by `RTW.BuildInfo`.

`linkobjs`

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

The function removes duplicate link objects that

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

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

`paths`

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

`priority` (optional)

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

`precompiled` (optional)

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

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

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

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

`linkonly` (optional)

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

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

If *`linkonly`* is `false` (the default), rules for building the link objects are added to the makefile. In this case, the value of *`precompiled`* determines which subsection of the added rules is expanded, `START_PRECOMP_LIBRARIES` (`true`) or `START_EXPAND_LIBRARIES` (`false`). The software performs the expansion of the `START_PRECOMP_LIBRARIES` or `START_EXPAND_LIBRARIES` macro only if your code generation target uses the template makefile approach for building code.

`groups` (optional)

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

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

You can apply

- A single group name to a linkable object

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

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

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

Description

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

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

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

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

If You Specify *priority*, *precompiled*, or *linkonly* as a...	The Function...
Value	Applies the value to objects it adds to the build information.

If You Specify *priority*, *precompiled*, or *linkonly* as a...	The Function...
Vector of values	Pairs each value with a specified object. Thus, the length of the vector must match the length of the cell array you specify for *linkobjs*.

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

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

Examples

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

Introduced in R2006a

addNonBuildFiles

Add nonbuild-related files to model build information

Syntax

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

paths and *groups* are optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

filenames

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

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

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

The function removes duplicate nonbuild file entries that

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

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

paths (optional)

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

groups (optional)

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

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

You can apply

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

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

Description

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

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

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

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

Examples

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`getNonBuildFiles`

Introduced in R2008a

addSourceFiles

Add source files to model build information

Syntax

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

paths and *groups* are optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

filenames

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

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

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

The function removes duplicate source file entries that

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

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

paths (optional)

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

groups (optional)

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

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

You can apply

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

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

Description

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

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

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to source files it adds to the build information.

If You Specify an Optional Argument as a...	The Function...
Cell array of character arrays	Pairs each character array with a specified source file. Thus, the length of the cell array must match the length of the cell array you specify for *`filenames`*.

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

Examples

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

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

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

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

- Add the source files `test1.c`, `test2.c`, and `driver.c` to build information `myModelBuildInfo`. Group the files `test1.c` and `test2.c` with the character vector `Tests` and 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 the `.c` files in a specified folder to build information `myModelBuildInfo` and place the files in the group `CFiles`.

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

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

Introduced in R2006a

addSourcePaths

Add source paths to model build information

Syntax

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

groups is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

paths

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

The function removes duplicate source file entries that

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

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

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

groups (optional)

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

- Document the use of specific source paths

- Retrieve or apply groups of source paths

You can apply

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

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

Description

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

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

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

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

Examples

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

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

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

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

- Add the source paths `/etcproj/etclib`, `/etcproj/etc/etc_build`, and `/common/lib` to build information `myModelBuildInfo`. Group the paths `/etc/proj/etclib` and `/etcproj/etc/etc_build` with the 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'});
  ```

More About

- "Customize Post-Code-Generation Build Processing"

See Also

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

Introduced in R2006a

addTMFTokens

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

Syntax

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

`groups` is optional.

Arguments

`buildinfo`

Build information returned by `RTW.BuildInfo`.

`tokennames`

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

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

`tokenvalues`

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

`groups` (optional)

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

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

You can apply

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

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

Description

Call the `addTMFTokens` function inside a post code generation command to provide build-time information to help customize makefile generation. The tokens specified in the `addTMFTokens` function call must be handled in the template makefile (TMF) for the target selected for your model. For example, if your post code generation command calls `addTMFTokens` to add a TMF token named `|>CUSTOM_OUTNAME<|` that specifies an output file name for the build, the TMF must take action with the value of `|>CUSTOM_OUTNAME<|` to achieve the desired result. (See "Examples" on page 2-36.)

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.

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

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

Examples

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

Introduced in R2009b

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()
```

More About

- "Reports for Code Generation"

See Also

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

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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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

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

Navigation

`'IncludeHyperlinkInReport'` — Code-to-model hyperlinks

'off' | 'on'

Code-to-model hyperlinks, specified as 'on' or 'off'. Specify 'on' to include code-to-model hyperlinks in the code generation report. The hyperlinks link code to the corresponding blocks, Stateflow® objects, and MATLAB® functions in the model diagram. For more information see "Code-to-model" on page 10-13.

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 10-15.

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 5-9.

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 10-18.

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 10-20.

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 10-22.

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 10-24.

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 5-11.

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

Data Types: `char`

More About

- "Reports for Code Generation"
- "Generate a Code Generation Report"
- "Generate Code Generation Report After Build Process"

See Also

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

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

More About

- "Reports for Code Generation"
- "Open Code Generation Report"

See Also

coder.report.close | coder.report.generate

Introduced in R2012a

findBuildArg

Search for a specific build argument in model build information

Syntax

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

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

buildArgName

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

Output Arguments

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

Vector	Description
identifier	Name of the build argument that the function finds
value	Value of the build argument

Description

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

Examples

- Find a build argument and its value stored in build information `myModelBuildInfo`.

```
load buildInfo.mat

myModelBuildInfo = buildInfo;

myBuildArgExtmodeStaticAlloc = 'EXTMODE_STATIC_ALLOC';

[buildArgId buildArgValue]=findBuildArg(buildInfo,myBuildArgExtmodeStaticAlloc);
```

View the argument identifier and value:

```
>> buildArgId

buildArgId =

EXTMODE_STATIC_ALLOC

>> buildArgValue

buildArgValue =

0
```

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`getBuildArgs`

Introduced in R2014a

findIncludeFiles

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

Syntax

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

`extPatterns` is optional.

Arguments

`buildinfo`

Build information returned by `RTW.BuildInfo`.

`extPatterns` (optional)

A cell array of character arrays that specify patterns of file name extensions for which the function is to search. Each pattern

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

The default pattern is `*.h`.

Examples of valid patterns include

```
*.h
*.hpp
*.x*
```

Description

The `findIncludeFiles` function

- Searches for include files, based on specified file name extension patterns, in source and include paths recorded in the model build information object
- Adds the files found, along with their full paths, to the build information object

- Deletes duplicate entries

Examples

Find include files with filename extension `.h` that are in build information object `myModelBuildInfo`, and add the full paths for the files found to the object.

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addIncludeFiles` | `packNGo` | `getIncludeFiles`

Introduced in R2006b

getBuildArgs

Build arguments from model build information

Syntax

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

includeGroupIDs and *excludeGroupIDs* are optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

includeGroupIDs (optional)

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

excludeGroupIDs (optional)

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

Output Arguments

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

Vector	Description
identifiers	Names of the build arguments
values	Values of the build arguments

Description

The `getBuildArgs` function returns build arguments stored in the model build information. Using optional *includeGroupIDs* and *excludeGroupIDs* arguments,

you can selectively include or exclude groups from the build arguments returned by the function.

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

Examples

- Get the build arguments stored in build information `myModelBuildInfo`.

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

  View the argument identifiers and values:

  ```
  >> 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'
      'MODELREF_LINK_LIBS'
      'RELATIVE_PATH_TO_ANCHOR'
      'MODELREF_TARGET_TYPE'
      'ISPROTECTINGMODEL'

  >> buildArgValues
  ```

```
buildArgValues =

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

More About

- “Customize Post-Code-Generation Build Processing”

See Also

`findBuildArg`

Introduced in R2014a

getCompileFlags

Compiler options from model build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

includeGroups (optional)

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

excludeGroups (optional)

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

Output Arguments

Compiler options stored in the model build information.

Description

The `getCompileFlags` function returns compiler options stored in the model build information. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of options the function returns.

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

Examples

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

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

compflags =

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

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

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

compflags =

    '-Zi -Wall'
```

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

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

compflags =

    '-O3'
```

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addCompileFlags` | `getDefines` | `getLinkFlags`

Introduced in R2006a

getDefines

Preprocessor macro definitions from model build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

includeGroups (optional)

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

excludeGroups (optional)

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

Output Arguments

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

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

Description

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

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

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

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

Examples

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

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

defs =

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

names

names =

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

values
```

```
values =

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

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

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

defs =

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

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

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

defs =

    '-DPRODUCTION'
```

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addDefines` | `getCompileFlags` | `getLinkFlags`

Introduced in R2006a

getFullFileList

List of files from model build information

Syntax

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

fcase is optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

fcase (optional)

The character vector `'source'`, `'include'`, or `'nonbuild'`. If the argument is omitted, the function returns files from the model build information.

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

Output Arguments

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

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

`updateFilePathsAndExtensions` to resolve file paths and extensions before calling `getFullFileList`.

Description

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

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

Examples

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 descendents (submodels).

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

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 descendents or submodels).

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

More About

- "Customize Post-Code-Generation Build Processing"

Introduced in R2008a

getIncludeFiles

Include files from model build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

concatenatePaths

The logical value `true` or `false`.

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

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

replaceMatlabroot

The logical value `true` or `false`.

If You Specify...	The Function...
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.

If You Specify...	The Function...
`false`	Does not replace the token `$(MATLAB_ROOT)`.

`includeGroups` (optional)

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

`excludeGroups` (optional)

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

Output Arguments

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

Description

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

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

Examples

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

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

```
incfiles =

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

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

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

incfiles =

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

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

Introduced in R2006a

getIncludePaths

Include paths from model build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

replaceMatlabroot

The logical value `true` or `false`.

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

includeGroups (optional)

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

excludeGroups (optional)

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

Output Arguments

Paths of include files stored in the model build information.

Description

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

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

Examples

- Get the include paths stored in 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 the paths in group `shared` that are stored in 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''
```

More About

- “Customize Post-Code-Generation Build Processing”

See Also

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

Introduced in R2006a

getLinkFlags

Link options from model build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by RTW.BuildInfo.

includeGroups (optional)

A character array or cell array that specifies groups of linker flags you want the function to return.

excludeGroups (optional)

A character array or cell array that specifies groups of linker flags you do not want the function to return. To exclude groups and not include specific groups, specify an empty cell array ('') for *includeGroups*.

Output Arguments

Linker options stored in the model build information.

Description

The getLinkFlags function returns linker options stored in the model build information. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of options the function returns.

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

Examples

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

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

linkflags =

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

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

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

linkflags =

    '-MD -Gy'
```

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

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

linkflags =

    '-T'
```

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addLinkFlags` | `getCompileFlags` | `getDefines`

Introduced in R2006a

getNonBuildFiles

Nonbuild-related files from model build information

Syntax

```
files=getNonBuildFiles(buildinfo, concatenatePaths, replaceMatlabroot, includeGroups, e
```

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

concatenatePaths

The logical value `true` or `false`.

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

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

replaceMatlabroot

The logical value `true` or `false`.

If You Specify...	The Function...
`true`	Replaces the token `$(MATLAB_ROOT)` with the absolute path for your MATLAB installation folder.

If You Specify...	The Function...
`false`	Does not replace the token `$(MATLAB_ROOT)`.

`includeGroups` (optional)

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

`excludeGroups` (optional)

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

Output Arguments

Names of nonbuild files stored in the model build information.

Description

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

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

Examples

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

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

nonbuildfiles =
```

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addNonBuildFiles`

Introduced in R2008a

getSourceFiles

Source files from model build information

Syntax

srcfiles=getSourceFiles(*buildinfo*, *concatenatePaths*, *replaceMatlabroot*, *includeGroups*,

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

concatenatePaths

The logical value `true` or `false`.

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

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

replaceMatlabroot

The logical value `true` or `false`.

If You Specify...	The Function...
`true`	Replaces path tokens, such as `$(MATLAB_ROOT)` and `$(START_DIR)`, with the absolute path.

If You Specify...	The Function...
`false`	Does not replace path tokens with the absolute path.

`includeGroups` (optional)

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

`excludeGroups` (optional)

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

Output Arguments

Names of source files stored in the model build information.

Description

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

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

Examples

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

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

```
srcfiles =

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

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

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

incfiles =

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addSourceFiles` | `getIncludeFiles` | `getIncludePaths` | `getSourcePaths` | `updateFilePathsAndExtensions`

Introduced in R2006a

getSourcePaths

Source paths from model build information

Syntax

files=getSourcePaths(*buildinfo*, *replaceMatlabroot*, *includeGroups*, *excludeGroups*)

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo

Build information returned by RTW.BuildInfo.

replaceMatlabroot

The logical value true or false.

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

includeGroups (optional)

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

excludeGroups (optional)

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

Output Arguments

Paths of source files stored in the model build information.

Description

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

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

Examples

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

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

srcpaths =

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

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

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

srcpaths =

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

- Get a path stored in build information `myModelBuildInfo`. First get the path without replacing `$(MATLAB_ROOT)` with an absolute path, then get it with

replacement. The MATLAB root folder in this case is `\\myserver\myworkspace\matlab`.

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

ans =

$(MATLAB_ROOT)\rtw\c\src

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

ans =

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

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

Introduced in R2006a

model_initialize

Initialization entry point in generated code for Simulink model

Syntax

```
void model_initialize(void)
```

Calling Interfaces

The calling interface generated for this function differs depending on the value of the model parameter **Code interface packaging** on page 9-24:

- `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** option to control whether an allocation function is generated.

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

Description

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

More About

- "Entry-Point Functions and Scheduling"

See Also

model_step | model_terminate

Introduced before R2006a

model_step

Step routine entry point in generated code for Simulink model

Syntax

```
void model_step(void)

void model_stepN(void)
```

Calling Interfaces

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

Parameter Value	Function Prototype
Off (single rate or multirate)	`void model_step(void);`
On (multirate)	`void model_stepN (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 9-24:

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

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

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

Note: If you have an Embedded Coder license:

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

Description

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

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

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

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

For 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. See “rt_OneStep and Scheduling Considerations” in the Embedded Coder documentation for a description of how calls to *model*_step are generated and scheduled.

Note: 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

---

More About

- “Entry-Point Functions and Scheduling”

See Also

model_initialize | model_terminate

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 9-24:

- `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.

More About

- "Entry-Point Functions and Scheduling"

See Also

`model_initialize` | `model_step`

Introduced before R2006a

packNGo

Package model code in zip file for relocation

Syntax

`packNGo(buildinfo, propVals...)`

`propVals` is optional.

Arguments

`buildinfo`

Build information returned by `RTW.BuildInfo`.

`propVals` (optional)

A cell array of property-value pairs that specify packaging details.

To...	Specify Property...	With Value...
Package model code files in a zip file as a single, flat folder.	`'packType'`	`'flat'` (default)
Package model code files hierarchically in a primary zip file. The value of the `'nestedZipFiles'` property determines whether the primary zip file contains secondary zip files or folders.	`'packType'`	`'hierarchical'`
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 build folder	`'nestedZipFiles'`	`true` (default)

To...	Specify Property...	With Value...
• `otherFiles.zip` — required files not in the *`matlabroot`* or `start` folder trees Paths for files in the secondary zip files are relative to the root folder of the primary zip file.		
Create a primary zip file that contains folders, for example, your build folder and *`matlabroot`*.	`'nestedZipFiles'`	`false`
Specify a file name for the primary zip file.	`'fileName'`	*`'text'`* Default:`'`*`model`*`.zip'` If you omit the `.zip` file extension, the function adds it for you.
Include only the minimal header files required to build the code in the zip file.	`'minimalHeaders'`	`true` (default)
Include header files found on the include path in the zip file.	`'minimalHeaders'`	`false`
Include the `html` folder for your code generation report.	`'includeReport'`	`true` (default is `false`)
Direct `packNGo` not to error out on parse errors.	`'ignoreParseError'`	`true` (default is `false`)
Direct `packNGo` not to error out if files are missing.	`'ignoreFileMissing'`	`true` (default is `false`)

Description

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

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` and `.hpp` files)

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

You might use this function to relocate files so they can be recompiled for a specific target environment or rebuilt in a development environment in which MATLAB is not installed.

By default, the function packages the files as a flat folder structure in a zip file named *`model`*`.zip`. You can tailor the output by specifying property name and value pairs as explained above.

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

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

Examples

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

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

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

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

Alternatives

You can configure model code packaging by selecting the **Package code and artifacts** on page 4-40 option on the **Code Generation** pane of the Configuration Parameters dialog box.

More About

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

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`
`dataTypeID`	An internal data type identifier

Field	Description
`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]
```

More About

- "Create a MAT-File That Includes a Model Parameter Structure"
- "Update Diagram and Run Simulation"
- "Default parameter behavior"
- "Block Creation"
- "Tune Parameters"

See Also

`rsimsetrtpparam`

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.

`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

Introduced in R2009b

rtw_precompile_libs

Build libraries within model without building model

Syntax

```
rtw_precompile_libs('model', build_spec)
```

Description

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

Input Arguments

`model`

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

`build_spec`

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

Field	Value
`rtwmakecfgDirs`	Cell array of character vectors that names the folders containing `rtwmakecfg` files for libraries that you want to precompile. 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

Field	Value
	makefile that the build approach generates contains the library rules only if the conversion requires the libraries.
`libSuffix` (optional)	Character vector that specifies the suffix, including the file type extension, to append to the name of each library (for example, `.a` or `_vc.lib`). The character vector must include a period (.). Set the suffix with either this field or the `TargetLibSuffix` parameter. If you specify a suffix with both mechanisms, the `TargetLibSuffix` setting overrides the setting of this field.
`intOnlyBuild` (optional)	Boolean flag. When set to `true`, indicates the function optimizes the libraries so that they compile from integer code only. Applies to ERT-based targets only.
`makeOpts` (optional)	Character vector that specifies an option to include in the `rtwMake` command line.
`addLibs` (optional)	Cell array of structures that specify 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.

Examples

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

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

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

% Define a build specification that specifies the location of the files to compile.
build_spec = [];
```

```
build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};

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

More About

- “Use rtwmakecfg.m API to Customize Generated Makefiles”
- “Precompile S-Function Libraries”
- “Recompile Precompiled Libraries”
- “Configure Build Process”

Introduced in R2009b

rtwbuild

Initiate build process

Syntax

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

rtwbuild(subsystem)

rtwbuild(subsystem,'Mode','ExportFunctionCalls')
blockHandle = rtwbuild(subsystem,'Mode','ExportFunctionCalls')
rtwbuild(subsystem,'Mode','ExportFunctionCalls,
'ExportFunctionInitializeFunctionName', 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 time for code generation, when rebuilding a model, `rtwbuild` provides incremental model build, which only rebuilds a model or sub-models that have changed since the most recent model build. To force a top-model build, see the `'ForceTopModelBuild'` argument.

Note: Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (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')`, if you have an Embedded Coder software license, generates code from `subsystem` that includes function calls that you can export to external application code.

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

`rtwbuild(subsystem,'Mode','ExportFunctionCalls', 'ExportFunctionInitializeFunctionName', 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_pri rtwdemo_rtwintrotype	rtGetInf.c rtGetInf.h rtGetNaN.c rtGetNaN.h rt_nonfinite.c rt_nonfinite.h	rtmodel.h	rt_logging.c

Model Files	Shared Files	Interface Files	Other Files

	`rtwtypes.h`
	`multiword_types.`
	`builtin_typeid_t`		

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 appears, providing information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Force Top Model Build

Generate code and build an executable image for `rtwdemo_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 Amplifier.h Amplifier_priva Amplifier_types	rtGetInf.c rtGetInf.h rtGetNaN.c rtGetNaN.h rt_nonfinite.c rt_nonfinite.h rtwtypes.h multiword_types.h builtin_typeid_typ	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 `Amplifier.exe`
Code Generation > Report > Create code generation report is selected	Report appears, providing information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores

Build Subsystem for Exporting Code to External Application

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

```
rtwdemo_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** > **All 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`.

Input Arguments

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

handle | name

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

Example: `'rtwdemo_exporting_functions'`

`subsystem` — Subsystem for which to generate code or build executable image

name

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

Example: `'rtwdemo_exporting_functions/rtwdemo_subsystem'`

Data Types: `char`

Name-Value Pair Arguments

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

Example: `rtwbuild('rtwdemo_mdlreftop','ForceTopModelBuild',true)`

`'ForceTopModelBuild'` — Force regeneration of top model code

false (default) | true

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

If You Want to...	Specify...
Force the code generator to regenerate code for the top model of a system that includes referenced models	`true`
Let 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 make changes associated with external or custom code, such as code for a custom target. For example, you should set `ForceTopModelBuild` to `true` if you change

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

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

`'OkayToPushNags'` — Display build error messages in Diagnostic Viewer

false (default) | true

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

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

If You Want to...	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`.

If You Want to...	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`

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** > **All ParametersCreate block** is set to `SIL`.

More About

Tips

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

- Press **Ctrl+B**.
- Select **Code** > **C/C++ Code** > **Build Model**.
- Invoke the `slbuild` command from the MATLAB command line.
- Initiate the Build Process

- “Configure Build Process”
- Control Regeneration of Top Model Code
- “Export Function-Call Subsystems”
- “Software-in-the-Loop Simulation”

See Also

`rtwrebuild` | `slbuild`

Introduced in R2009a

RTW.getBuildDir

Build folder information for model

Syntax

```
RTW.getBuildDir(model)
folderstruct = RTW.getBuildDir(model)
```

Description

`RTW.getBuildDir(model)` displays build folder information for model.

If the model is closed, the function opens and then closes the model, leaving it in its original state. If the model is large and closed, the `RTW.getBuildDir` function can take significantly longer to execute.

`folderstruct = RTW.getBuildDir(model)` returns a structure containing build folder information.

You can use this function in automated scripts to determine the build folder in which the generated code for a model is placed.

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

Examples

Display Build Folder Information

```
>> RTW.getBuildDir('sldemo_fuelsys')


ans =

              BuildDirectory: 'C:\work\modelref\sldemo_fuelsys_ert_rtw'
                 CacheFolder: 'C:\work\modelref'
               CodeGenFolder: 'C:\work\modelref'
```

```
          RelativeBuildDir: 'sldemo_fuelsys_ert_rtw'
            BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\sldemo_fuelsys'
      ModelRefRelativeSimDir: 'slprj\sim\sldemo_fuelsys'
      ModelRefRelativeHdlDir: 'slprj\hdl\sldemo_fuelsys'
           ModelRefDirSuffix: ''
           SharedUtilsSimDir: 'slprj\sim\_sharedutils'
           SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

Get Build Folder Information

Return build folder information for the model MyModel.

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


folderstruct =

              BuildDirectory: 'H:\MyModel_ert_rtw'
                 CacheFolder: 'H:\'
               CodeGenFolder: 'H:\'
            RelativeBuildDir: 'MyModel_ert_rtw'
              BuildDirSuffix: '_ert_rtw'
  ModelRefRelativeRootSimDir: 'slprj\sim'
  ModelRefRelativeRootTgtDir: 'slprj\ert'
    ModelRefRelativeBuildDir: 'slprj\ert\MyModel'
      ModelRefRelativeSimDir: 'slprj\sim\MyModel'
      ModelRefRelativeHdlDir: 'slprj\hdl\MyModel'
           ModelRefDirSuffix: ''
           SharedUtilsSimDir: 'slprj\sim\_sharedutils'
           SharedUtilsTgtDir: 'slprj\ert\_sharedutils'
```

Input Arguments

`model` — Input data

character vector

Character vector specifying the name of a Simulink model.

Example: `'sldemo_fuelsys'`

Data Types: `char`

Output Arguments

`folderstruct` — Output data

structure

Structure containing the following:

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

More About

- "Working Folder"
- "Folders Used by the Build Process"
- "Control the Location for Generated Files"

See Also

`rtwbuild`

Introduced in R2008b

rtwrebuild

Rebuild generated code

Syntax

```
rtwrebuild()

rtwrebuild('model')

rtwrebuild('path')
```

Description

`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.

`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('model')` assumes that the current working folder is one level above the build folder (as indicated by `pwd` when initiating the model build) and invokes the makefile in the build folder. If the current working folder is not one level above the build folder, the function exits with an error.

`rtwrebuild('path')` finds the build folder indicated with the path and invokes the makefile in the build folder. This syntax lets the function operate without regard to the relationship between the current working folder and the build folder of the model.

Note: Do not use `rtwbuild`, `rtwrebuild`, or `slbuild` commands with parallel language features (for example, within a `parfor` or `spmd` loop). For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models”.

Input Arguments

`model` — Character vector specifying the model name

Example: `'mymodel'`

`path` — Character vector specifying the fully qualified path to the build folder for the model

Example: `fullfile('C:','work','mymodel_grt_rtw')`

Examples

Rebuild Code from Build Folder

Invoke the makefile and recompile code when the current working folder is the build folder. If the model name is `mymodel`, the model build was initiated in the `C:\work` folder, and 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

Invoke the makefile and recompile code when the current working folder is one level above the build folder.

```
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'))
```

More About

- “Rebuild a Model”

See Also

`rtwbuild` | `slbuild`

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. 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`

More About

- "Working with the Report Explorer"
- Code Generation Summary

Introduced in R2007a

rtwtrace

Trace a block to generated code in HTML code generation report

Syntax

```
rtwtrace('blockpath')
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:

- You select an ERT-based model and enabled model to code navigation.

  To do this, on the Configuration Parameters dialog box, select the **All Parameters** tab, and select the **Model-to-code** parameter.
- You generate code for the model using the code generator.
- You have the build folder under the current working folder; otherwise, `rtwtrace` may produce an error.

`rtwtrace('blockpath', 'hdl')` opens an HTML code generation report for HDL Coder™ that displays contents of the source code file, and highlights the line of code corresponding to the specified block.

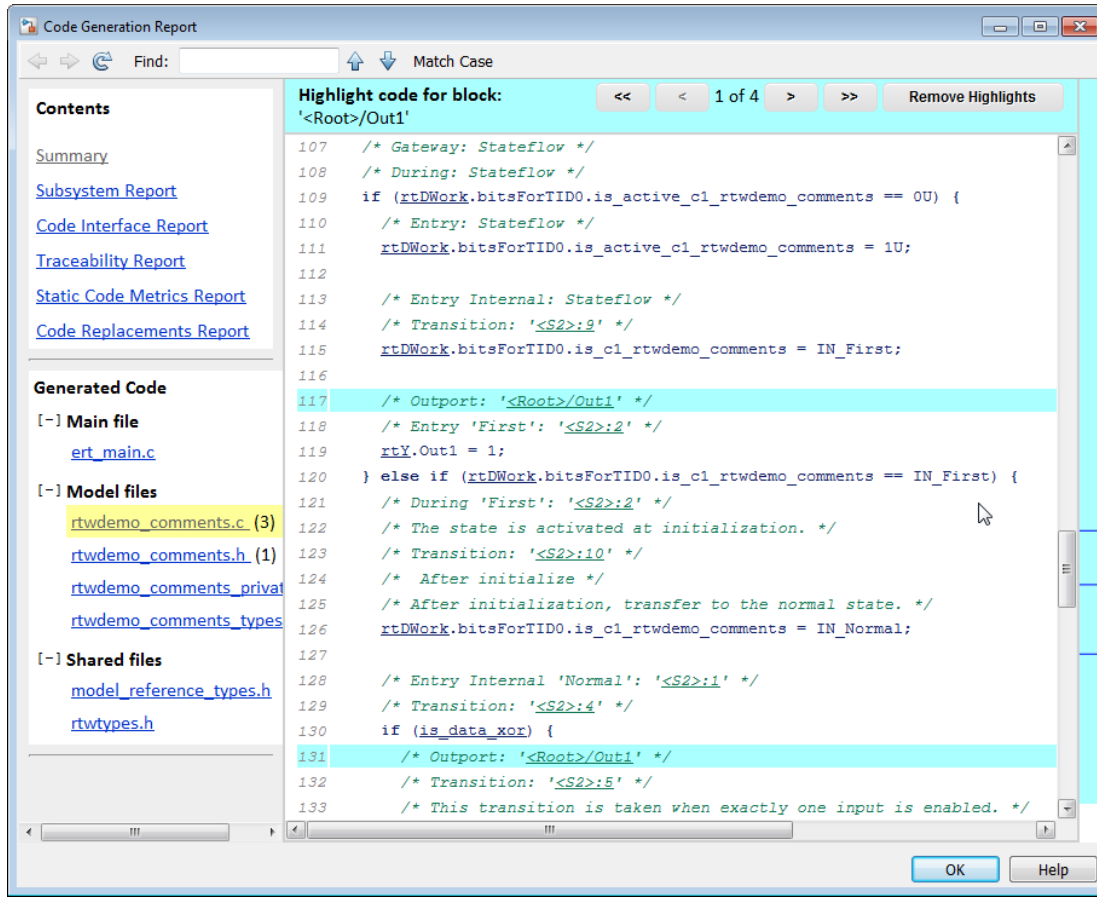
`rtwtrace('blockpath', 'plc')` opens an HTML code generation report for Simulink PLC Coder™ that displays contents of the source code file, and highlights the line of code corresponding to the specified block.

Examples

Display Generated Code for a Block

Display the generated code for block `Out1` in the model `rtwdemo_comments` in HTML code generation report:

```
rtwtrace('rtwdemo_comments/Out1')
```



Input Arguments

`blockpath` — block path

character vector (default)

`blockpath` is a character vector enclosed in quotes specifying the full Simulink block path, for example, '*`model_name`*/*`block_name`*'.

Example: `'Out1'`

Data Types: `char`

`hdl` — HDL Coder

character vector

`hdl` is a character vector enclosed in quotes specifying 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 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**.

Introduced in R2009b

setTargetProvidesMain

Disable inclusion of a code generator provided (generated or static) `main.c` source file

Syntax

```
setTargetProvidesMain(buildinfo,varargin)
```

Description

`setTargetProvidesMain(buildinfo,varargin)` can appear in the `'after_tlc'` case in the `ert_make_rtw_hook.m` or `grt_make_rtw_hook.m` file. Use the `setTargetProvidesMain` function to disable the build process from generating a sample `main.obj` object file.

Input Arguments

`buildinfo`

Build information returned by `RTW.BuildInfo`.

`varargin`

A character array that specifies whether the target provides `main.c` or the build process generates a sample `main.obj`.

- `false` — the build process generates a sample `main.obj` object file (default).
- `true` — the target provides the `main.c` source file.

Examples

Apply setTargetProvidesMain

To apply the `setTargetProvidesMain` function:

For ERT and ERT-derived code generation targets, set the **Configuration Parameters > Code Generation > Templates > Generated an example main program** parameter value to `'off'`.

Create a custom `ert_make_rtw_hook.m` or `grt_make_rtw_hook.m` file. See "Customize Build Process with STF_make_rtw_hook File".

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. The model requires this file to build without errors when you indicate that the target provides `main.c`.

More About

- "Customize Build Process with STF_make_rtw_hook File"

See Also

`addSourceFiles` | `addSourcePaths`

Introduced in R2009a

Simulink.fileGenControl

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

Syntax

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

Description

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

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

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

`Simulink.fileGenControl('setConfig', 'config', cfg, 'keepPreviousPath', true, 'createDir', true)` sets the file generation control

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

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

Naming Conflicts

Using `Simulink.fileGenControl` to set `CacheFolder` and `CodeGenFolder` adds the specified folders to your MATLAB search path. Be aware that there is the same potential for a naming conflict as when you use `addpath` to add folders to the search path. For example, a naming conflict occurs if the folder you specify for `CacheFolder` or `CodeGenFolder` contains a model file with the same name as an open model. For more information, see “What Is the MATLAB Search Path?” and “Files and Folders that MATLAB Accesses”.

Build artifacts (for example, a build folder present in the current working folder, selected cache folder, or selected code generation folder) from previous builds can contain conflicts if you change the default selection for the cache folder or code generation folder. To use a nondefault location (default location is the current working folder, `pwd`) for the cache folder or code generation folder, use these guidelines:

- Delete any potentially conflicting build artifacts from previous builds from the `pwd` folder, selected nondefault cache folder, and selected nondefault code generation folder.
- Change the cache folder selection or code generation folder selection with `Simulink.fileGenControl` or with Simulink preferences.

Input Arguments

`action`

Character vector specifying one of the following actions:

Action	Description
`getConfig`	Returns a handle to an instance of the `Simulink.FileGenConfig` object containing the current values of the `CacheFolder` and `CodeGenFolder` parameters.
`reset`	Reinitializes the `CacheFolder` and `CodeGenFolder` parameters to the values provided by the Simulink preferences “Simulation cache folder” and “Code generation folder”.
`set`	Sets the `CacheFolder` and/or `CodeGenFolder` parameters for the current MATLAB session by directly passing values.
`setConfig`	Sets the `CacheFolder` and/or `CodeGenFolder` parameters for the current MATLAB session by passing a handle to an instance of the `Simulink.FileGenConfig` object.

`'config', cfg`

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

`'CacheFolder', cacheFolderPath`

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

`'CodeGenFolder', codeGenFolderPath`

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

Note: You can specify absolute or relative paths to the build folders. For example:

- `'C:\Work\mymodelsimcache'` and `'/mywork/mymodelgencode'` specify absolute paths.
- `'mymodelsimcache'` is a path relative to the current working folder (`pwd`). The software converts a relative path to a fully qualified path at the time the `CacheFolder` or `CodeGenFolder` parameter is set. For example, if `pwd` is `'/mywork'`, the result is `'/mywork/mymodelsimcache'`.
- `'../test/mymodelgencode'` is a path relative to `pwd`. If `pwd` is `'/mywork'`, the result is `'/test/mymodelgencode'`.

`'keepPreviousPath', true`

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

`'createDir', true`

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

Output Arguments

`cfg`

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

Examples

Obtain the current `CacheFolder` and `CodeGenFolder` values:

```
cfg = Simulink.fileGenControl('getConfig');
myCacheFolder = cfg.CacheFolder;
myCodeGenFolder = cfg.CodeGenFolder;
```

Set the `CacheFolder` and `CodeGenFolder` parameters for the current MATLAB session by first setting fields in an instance of the `Simulink.FileGenConfig` object and then passing a handle to the object instance. 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');
Simulink.fileGenControl('setConfig', 'config', cfg);
```

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

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

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

More About

- “Control the Location for Generated Files”

See Also

“Code generation folder” | “Simulation cache folder”

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

character vector (default)

Model name, specified as a 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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true`
specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

General

`'Path'` — Folder for protected model

current working folder (default) | character vector

Folder for protected model, specified as a 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.

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

Functionality

`'Mode'` — Model protection mode

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

Model protection mode, specified as a character vector. 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, specified as a character vector. 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.protect` |
`Simulink.ModelReference.ProtectedModel.setPasswordForModify`

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`.

- Protected Models for Model Reference
- “Test the Protected Model”
- “Package a Protected Model”
- “Specify Custom Obfuscator for Protected Model”
- “Configure and Run SIL Simulation”
- “Define Callbacks for Protected Model”

Input Arguments

`model` — Model name

character vector (default)

Model name, specified as a 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 single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true`
specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

`'Harness'` — Option to create a harness model

false (default) | true

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

Example: `'Harness',true`

`'Mode'` — Model protection mode

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

Model protection mode, specified as a character vector. 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) | character vector

Folder for protected model, specified as a 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, specified as a character vector. 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.

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"

More About

- "Protected Model"
- "Protect a Referenced Model"
- "Protected Model File"
- "Harness Model"
- "Protected Model Report"
- "Code Generation Support in a Protected Model"
- "Code Interfaces for SIL and PIL"

See Also

`Simulink.ModelReference.modifyProtectedModel` | `Simulink.ModelReference.ProtectedModel.clearPasswords` | `Simulink.ModelReference.ProtectedModel.clearPasswordsForModel` | `Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` | `Simulink.ModelReference.ProtectedModel.setPasswordForModify` | `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` | `Simulink.ModelReference.ProtectedModel.setPasswordForView`

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()
```

More About

- "Protect a Referenced Model"

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.clearPasswordsFor

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

character vector

Model name specified as a character vector

Example: '`rtwdemo_counter`'

Data Types: `char`

More About

- "Protect a Referenced Model"

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords`

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.

`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 in the MATLAB documentation.

Examples

- "Specify Custom Obfuscator for Protected Model"

See Also

`Simulink.ModelReference.protect`

Simulink.ModelReference.ProtectedModel.setPasswordForCo

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

character vector

Model name, specified as a 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

character vector

Password, specified as a character vector. If the protected model is encrypted for code generation, the password is required.

See Also

`Simulink.ModelReference.protect` |
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` |
`Simulink.ModelReference.ProtectedModel.setPasswordForView`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForMo

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

character vector

Model name, specified as a 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

character vector

Password, specified as a 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.setPasswordForSim

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

character vector

Model name, specified as a 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

character vector

Password, specified as a character vector. If the protected model is encrypted for simulation, the password is required.

See Also

`Simulink.ModelReference.protect` |
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForView`

Introduced in R2014b

Simulink.ModelReference.ProtectedModel.setPasswordForVie

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

character vector

Model name, specified as a 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

character vector

Password, specified as a character vector. If the protected model is encrypted for read-only view, the password is required.

See Also

`Simulink.ModelReference.protect` |
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` |
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`

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')
```

- "Create a Protected Model with Multiple Targets"

Input Arguments

`model` — Model name

character vector

Model name, specified as a 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
```

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 2-134 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, specified as a character vector. 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, specified as a character vector. 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

character vector

MATLAB commands to execute in response to an event, specified as a character vector.

`callbackFile` — Callback script to execute

character vector

MATLAB script to execute in response to an event, specified as a 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, specified as a character vector. 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

character vector

MATLAB script to execute in response to an event, specified as a character vector. Script must be on the MATLAB path.

Example: `'pmCallback.m'`

`CallbackText` — Callback code to execute

character vector

MATLAB commands to execute in response to an event, specified as a character vector.

Example: `'A = [15 150];disp(A)'`

`Event` — Event that triggers callback

`'PreAccess' | 'Build'`

Callback trigger event, specified as a character vector. 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 in the MATLAB documentation.

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`

More About

- "Protect a Referenced Model"
- "Code Generation Support in a Protected Model"

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

More About

- “Protect a Referenced Model”
- “Code Generation Support in a Protected Model”

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, specified as a character vector.

`Event` — Event that triggered callback

`'PreAccess' | 'Build'`

Callback trigger event, specified as a character vector. 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, specified as a character vector. 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 in the MATLAB documentation.

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`

More About

- “Protect a Referenced Model”

- “Code Generation Support in a Protected Model”

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

character vector

Protected model name, specified as a character vector.

`event` — Event that triggered callback

`'PreAccess' | 'Build'`

Callback trigger event, specified as a character vector. 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, specified as a character vector. 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.

More About

- "Protect a Referenced Model"
- "Code Generation Support in a Protected Model"

See Also

`Simulink.ModelReference.protect` | Simulink.ProtectedModel.CallbackInfo

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

character vector

Model name, specified as a 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

More About

- "Protect a Referenced Model"
- "Code Generation Support in a Protected Model"

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')
```

- “Create a Protected Model with Multiple Targets”

- “Use a Protected Model with Multiple Targets”

Input Arguments

`protectedModel` — Model name

character vector

Protected model name, specified as a character vector.

`targetID` — Target identifier

character vector

Identifier for selected target, specified as a 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`

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')
```

- "Create a Protected Model with Multiple Targets"
- "Use a Protected Model with Multiple Targets"

Input Arguments

`protectedModel` — Model name

character vector

Protected model name, specified as a 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`

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')
```

- "Create a Protected Model with Multiple Targets"
- "Use a Protected Model with Multiple Targets"

Input Arguments

`protectedModel` — Model name

character vector

Protected model name, specified as a 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`

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

character vector

Protected model name, specified as a character vector.

`type` — Open method

'webview' | 'report'

Method for viewing the protected model, specified as a character vector. 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')
```

- "Create a Protected Model with Multiple Targets"

Input Arguments

`protectedModel` — Model name

character vector

Protected model name, specified as a character vector.

`targetID` — Target to be removed

character vector

Identifier for target to be removed, specified as a 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

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')
```

- “Create a Protected Model with Multiple Targets”
- “Use a Protected Model with Multiple Targets”

Input Arguments

`protectedModel` — Model name

character vector

Protected model name, specified as a character vector.

`targetID` — Target identifier

character vector

Identifier for selected target, specified as a character vector.

See Also

`Simulink.ModelReference.protect` | `Simulink.ProtectedModel.addTarget` | `Simulink.ProtectedModel.getConfigSet` | `Simulink.ProtectedModel.getCurrentTarget` | `Simulink.ProtectedModel.getSupportedTargets` | `Simulink.ProtectedModel.removeTarget`

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 **Terminate function required** option on the **All Parameters** tab of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg,hSrc)

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

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

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

More About

- "Define and Display Custom Target Options"
- "Custom Target Optional Features"

See Also

`slConfigUISetEnabled` | `slConfigUISetVal`

Introduced in R2012a

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 **Terminate function required** option on the **All Parameters** tab of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg,hSrc)

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

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

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

More About

- "Define and Display Custom Target Options"
- "Custom Target Optional Features"

See Also

`slConfigUIGetVal` | `slConfigUISetVal`

Introduced in R2012a

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 **Terminate function required** option on the **All Parameters** tab of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg,hSrc)

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

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

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

More About

- "Define and Display Custom Target Options"
- "Custom Target Optional Features"

See Also

`slConfigUIGetVal` | `slConfigUISetEnabled`

Introduced in R2012a

switchTarget

Select target for configuration set

Syntax

```
switchTarget(myConfigObj,systemTargetFile,[])
switchTarget(myConfigObj,systemTargetFile,targetOptions)
```

Description

`switchTarget(myConfigObj,systemTargetFile,[])` selects a system target file for the active configuration set.

`switchTarget(myConfigObj,systemTargetFile,targetOptions)` sets the configuration parameters specified by `targetOptions`.

Examples

Select target file without options

```
% Get the active configuration set for 'model'
myConfigObj = getActiveConfigSet(model);
% Change the system target file for the configuration set.
switchTarget(myConfigObj,'ert.tlc',[]);
```

Select target file with options

```
% Get the active configuration set for the current model
myConfigObj=getActiveConfigSet(gcs);

% Specify target options
targetOptions.TLCOptions = '-aVarName=1';
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = 'my target';
targetOptions.TemplateMakefile = 'grt_default_tmf';

% Verify values (optional)
targetOptions
```

```
          TLCOptions: '-aVarName=1'
         MakeCommand: 'make_rtw'
         Description: 'my target'
    TemplateMakefile: 'grt_default_tmf'

% Define a system target file
targetSystemFile='grt.tlc';

% Change the system target file and target options
% for the configuration set
switchTarget(myConfigObj,targetSystemFile,targetOptions);
```

Use options to select MSVC ERT target file, instead of default ERT target

```
% use switchTarget to select tmf build of MSVC ERT target
model='rtwdemo_rtwintro';
open_system(model);
myConfigObj = getActiveConfigSet(model);
targetOptions.MakeCommand = 'make_rtw';
targetOptions.Description = 'Create Visual C/C++ Solution File for Embedded Coder';
targetOptions.TemplateMakefile = 'RTW.MSVCBuild';
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Use options to select default ERT target file, instead of using set_param(model,'SystemTargetFile','ert.tlc')

```
% use switchTarget to select toolchain build of defaul ERT target
model='rtwdemo_rtwintro';
open_system(model);
myConfigObj = getActiveConfigSet(model);
targetOptions.MakeCommand = '';
targetOptions.Description = 'Embedded Coder';
targetOptions.TemplateMakefile = '';
switchTarget(myConfigObj,'ert.tlc',targetOptions);
```

Input Arguments

`myConfigObj` — Input data

configuration set object

A configuration set object of `Simulink.ConfigSet` or configuration reference object of `Simulink.ConfigSetRef`. Call `getActiveConfigSet` to get the configuration set object.

Example: `myConfigObj = getActiveConfigSet(model);`

`systemTargetFile` — Input data

name of system target file

Specify the name of the system target file, such as `ert.tlc` for Embedded Coder, or `grt.tlc` for Simulink Coder™.

Example: `systemTargetFile = 'ert.tlc';`

Data Types: `char`

`targetOptions` — Input options

structure of configuration parameter options

You can choose to modify certain configuration parameters by filling in values in a structure for fields listed below. If you do not want to use options, specify an empty structure([ ]).

Field	Value
`TemplateMakefile`	Character vector specifying file name of template makefile.
`TLCOptions`	Character vector specifying TLC argument.
`MakeCommand`	Character vector specifying make command MATLAB language file.
`Description`	Character vector specifying description of target.

Example: targetOptions.TemplateMakefile = 'grt_default_tmf';

Data Types: `struct`

More About

- "Select a System Target File Programmatically"
- "Select a System Target File"
- "Set Target Language Compiler Options"

See Also

`getActiveConfigSet` | `Simulink.ConfigSet` | `Simulink.ConfigSetRef`

Introduced in R2009b

tlc

Invoke Target Language Compiler to convert model description file to generated code

Syntax

```
tlc [-options] [file]
```

Description

`tlc` invokes the Target Language Compiler (TLC) from the command prompt. The TLC converts the model description file, *`model`*`.rtw` (or similar files), into target-specific code or text. Typically, you do not call this command because the build process automatically invokes the Target Language Compiler when generating code. For more information, see "Introduction to the Target Language Compiler".

Note: This command is used only when invoking the TLC separately from the build process. You cannot use this command to initiate code generation for a model.

```
tlc [-options] [file]
```

You can change the default behavior by specifying one or more compilation *options* as described in "Options" on page 2-203

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 Simulink Coder filename" on page 2-204
- "-v Specify verbose level" on page 2-204
- "-l Specify path to local include files" on page 2-204

-r Specify Simulink Coder filename

`-r` *file_name*

Specify the filename that you want to translate.

-v Specify verbose level

`-v` *number*

Specify a `number` indicating the verbose level. If you omit this option, the default value is one.

-l Specify path to local include files

`-l` *path*

Specify a folder `path` to local include files. The TLC searches this path in the order specified.

-m Specify maximum number of errors

`-m` *number*

Specify the maximum `number` of errors reported by the TLC prior to terminating the translation of the `.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

`-xO`

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)
```

extensions is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

extensions (optional)

A cell array of character arrays that specifies the extensions (file types) of files for which to search and include in the update processing. By default, the function searches for files with a `.c` extension. The function checks files and updates paths and extensions based on the order in which you list the extensions in the cell array. For example, if you specify `{'.c' '.cpp'}` and a folder contains `myfile.c` and `myfile.cpp`, an instance of `myfile` would be updated to `myfile.c`.

Description

Using paths that already exist in the model build information, the `updateFilePathsAndExtensions` function checks whether file references in the build information need to be updated with a path or file extension. This function can be particularly useful for

- Maintaining build information for a toolchain that requires the use of file extensions
- Updating multiple customized instances of build information for a given model

---

Note: If you need to use `updateFilePathsAndExtensions`, you should call it once, after you add files to the build information, to minimize the potential performance impact of the required disk I/O.

Examples

Create the folder path `etcproj/etc` in your working folder, add files `etc.c`, `test1.c`, and `test2.c` to the folder `etc`. This example assumes the working folder is `w:\work\BuildInfo`. From the working folder, update build information `myModelBuildInfo` with missing paths or file extensions.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, fullfile(pwd,...
 'etcproj', '/etc'), 'test');
addSourceFiles(myModelBuildInfo, {'etc' 'test1'...
 'test2'}, '', 'test');
before=getSourceFiles(myModelBuildInfo, true, true);
before

before =

    '\etc'    '\test1'    '\test2'

updateFilePathsAndExtensions(myModelBuildInfo);
after=getSourceFiles(myModelBuildInfo, true, true);
after{:}

ans =

w:\work\BuildInfo\etcproj\etc\etc.c


ans =

w:\work\BuildInfo\etcproj\etc\test1.c


ans =

w:\work\BuildInfo\etcproj\etc\test2.c
```

More About

- "Customize Post-Code-Generation Build Processing"

See Also

`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFileSeparator`

Introduced in R2006a

updateFileSeparator

Change file separator used in model build information

Syntax

```
updateFileSeparator(buildinfo, separator)
```

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

separator

A character array that specifies the file separator \ (Windows®) or / (UNIX®) to be applied to file path specifications.

Description

The `updateFileSeparator` function 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 makefile based 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 object `myModelBuildInfo` to apply the Windows file separator.

```
myModelBuildInfo = RTW.BuildInfo;
updateFileSeparator(myModelBuildInfo, '\');
```

More About

- “Customize Post-Code-Generation Build Processing”
- “Cross-Compile Code Generated on Microsoft Windows”

See Also

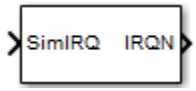
`addIncludeFiles` | `addIncludePaths` | `addSourceFiles` | `addSourcePaths` | `updateFilePathsAndExtensions`

Introduced in R2006a

Blocks — Alphabetical List

Async Interrupt

Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that are to execute downstream subsystems or Task Sync blocks



Library

Asynchronous / Interrupt Templates

Description

For each specified 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.

Parameters

VME interrupt number(s)

An array of VME interrupt numbers for the interrupts to be installed. The valid range is `1..7`.

The width of the Async Interrupt block output signal corresponds to the number of VME interrupt numbers specified.

Note A model can contain more than one Async Interrupt block. However, if you use more than one Async Interrupt block, do not duplicate the VME interrupt numbers specified in each block.

VME interrupt vector offset(s)

An array of unique interrupt vector offset numbers corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field. The Stateflow software passes the offsets to the RTOS (VxWorks) call `intConnect(INUM_TO_IVEC(offset),...)`.

Simulink task priority(s)

The Simulink priority of downstream blocks. Each output of the Async Interrupt block drives a downstream block (for example, a function-call subsystem). Specify an array of priorities corresponding to the VME interrupt numbers you specify for **VME interrupt number(s)**.

The **Simulink task priority** values are required to generate a rate transition code (see "Rate Transitions and Asynchronous Blocks"). Simulink task priority values are also required to maintain absolute time integrity when the asynchronous task needs to obtain real time from its base rate or its caller. The assigned priorities typically are higher than the priorities assigned to periodic tasks.

Note: The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

Preemption flag(s); preemptable-1; non-preemptable-0

The value 1 or 0. Set this option to 1 if an output signal of the Async Interrupt block drives a Task Sync block.

Higher priority interrupts can preempt lower priority interrupts in the example RTOS (VxWorks). To lock out interrupts during the execution of an ISR, set the preemption flag to `0`. This causes generation of `intLock()` and `intUnlock()` calls at the beginning and end of the ISR code. Use interrupt locking carefully, as it increases the system's interrupt response time for interrupts at the `intLockLevelSet()` level and below. Specify an array of flags corresponding to the VME interrupt numbers entered in the **VME interrupt number(s)** field.

Note The number of elements in the arrays specifying **VME interrupt vector offset(s)** and **Simulink task priority** must match the number of elements in the **VME interrupt number(s)** array.

Manage own timer

If checked, the ISR generated by the Async Interrupt block manages its own timer by reading absolute time from the hardware timer. Specify the size of the hardware timer with the **Timer size** option.

Timer resolution (seconds)

The resolution of the ISRs timer. ISRs generated by the Async Interrupt block maintain their own absolute time counters. By default, these timers obtain their values from the 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) might be different. You should 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), you should replace the `tickGet` call with an equivalent call to the target RTOS, or generate code to read the timer register on the target hardware. For more information, see "Timers in Asynchronous Tasks" and "Async Interrupt Block Implementation".

Timer size

The number of bits to be used to store the clock tick for a hardware timer. The ISR generated by the Async Interrupt block uses the timer size when you select **Manage own timer**. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the 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. However, when **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, the code generator uses a second 32-bit integer to address overflows.

For more information, see "Control Memory Allocation for Time Counters". See also "Timers in Asynchronous Tasks".

Enable simulation input

If checked, the Simulink software adds an input port to the Async Interrupt block. This port is for use in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Note: Before generating code, consider removing blocks that drive the simulation input to prevent the blocks from contributing to the generated code. Alternatively, you can use the Environment Controller block, as explained in "Dual-Model Approach: Code Generation". However, if you use the Environment Controller block, be aware that the sample times of driving blocks contribute to the sample times supported in the generated code.

Inputs and Outputs

Input

A simulated interrupt source.

Output

Control signal for a

- Function-call subsystem
- Task Sync block
- Stateflow chart configured for a function call input event

Assumptions and Limitations

- The block supports VME interrupts 1 through 7.
- The block uses the following 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. As a general rule, it is best to keep ISRs as short as possible. Connect only function-call subsystems that contain a small number of blocks to an Async Interrupt block.

A better solution for large subsystems is to use the Task Sync block to synchronize the execution of the function-call subsystem to a 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 for more information.

See Also

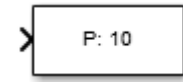
`Task Sync`
"Asynchronous Events"

Introduced in R2006a

Asynchronous Task Specification

Specify priority of asynchronous task represented by referenced model triggered by asynchronous interrupt

Library: / Asynchronous



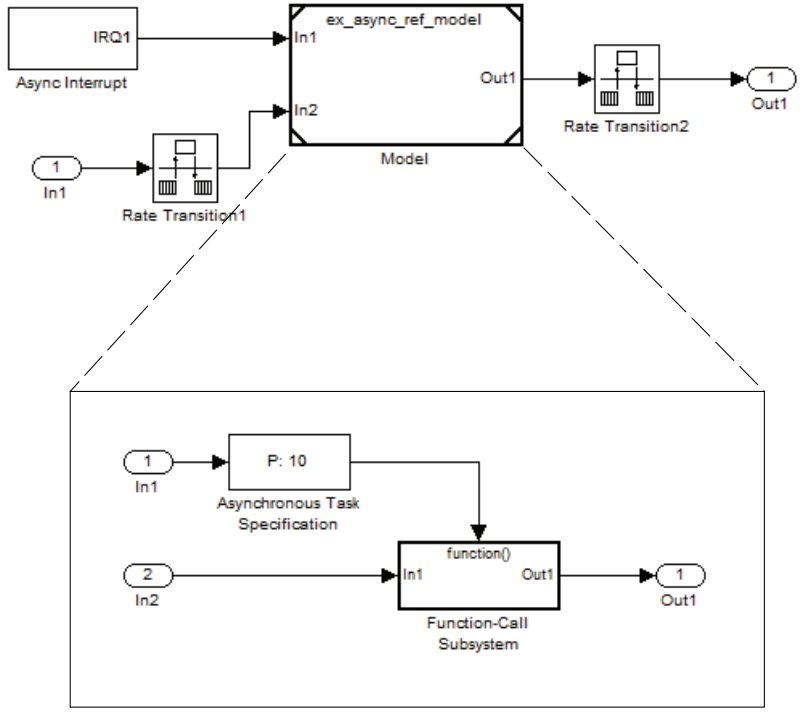
Description

The Asynchronous Task Specification block specifies parameters, such as the task priority, of an asynchronous task represented by a function-call subsystem that is triggered by an asynchronous interrupt. Use this block to control scheduling of function-call subsystems that are triggered by 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".

As the following figure shows:

- 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

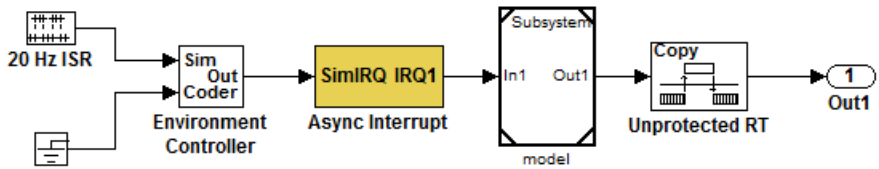
`Task priority` — Priority of asynchronous task that calls function-call subsystem

10 (default)

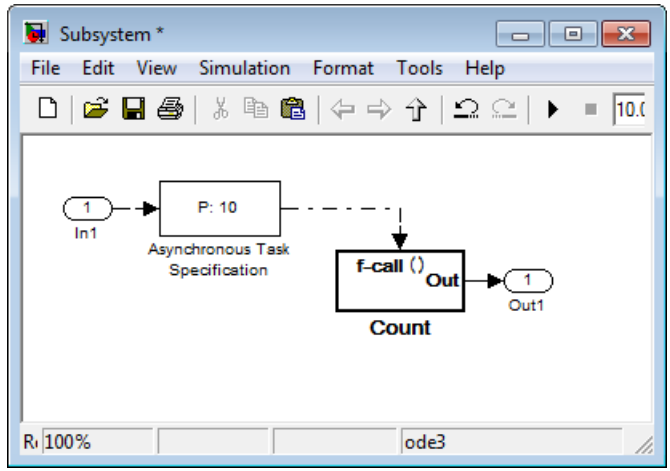
Specify an integer or [] as the priority of the asynchronous task that calls the connected function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

- 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. For this case, the rate transition algorithm is conservative (not optimized). The algorithm assumes that 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. Whereas, if the parameter is set to [], the priority of the Async Interrupt block can be a value other than 10.

More About

- "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 Referencing"

See Also

Blocks

`Function-Call Subsystem` | `Inport`

Introduced in R2011a

Generated S-Function

Represent model or subsystem as generated S-function code

Library

S-Function Target

Description

An instance of the Generated S-Function block represents code the 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, 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”.

- You must set the solver parameters of the Generated S-Function block to be the same as those of the original model or subsystem. The generated S-function code will operate identically to the original subsystem (for an exception to this rule, see Choice of Solver Type).

Parameters

Generated S-function name (model_sf)

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

If checked, displays modules generated for the S-function.

See Also

"Create S-Function Blocks from a Subsystem"

Introduced in R2011b

Model Header

Specify custom header code

Library

Custom Code

Description

The Model Header block adds user-specified custom code to the *model*`.h` file that the code generator creates for the model that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

Top of Model Header

Code to be added near the top of the generated model header file, in a `user code (top of header file)` section.

Bottom of Model Header

Code to be added at the bottom of the generated model header file, in a `user code (bottom of header file)` section.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

See Also

`Model Source`, `System Derivatives`, `System Disable`, `System Enable`, `System Initialize`, `System Outputs`, `System Start`, `System Terminate`, `System Update`
"Insert Code into Model Custom Code Blocks"

Introduced in R2006a

Model Source

Specify custom source code

Library

Custom Code

Description

The Model Source block adds user-specified custom code to the *`model`*`.c` or *`model`*`.cpp` file that the code generator creates for the model that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

Top of Model Source

Code to be added near the top of the generated model source file, in a `user code (top of source file)` section.

Bottom of Model Source

Code to be added at the bottom of the generated model source file, in a `user code (bottom of source file)` section.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

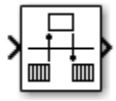
See Also

`Model Header`, `System Derivatives`, `System Disable`, `System Enable`, `System Initialize`, `System Outputs`, `System Start`, `System Terminate`, `System Update`
"Insert Code into Model Custom Code Blocks"

Introduced in R2006a

Protected RT

Handle transfer of data between blocks operating at different rates and maintain data integrity



Library

VxWorks (`vxlib1`)

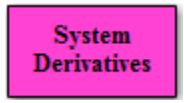
Description

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 custom system derivative code



Library

Custom Code

Description

The System Derivatives block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemDerivatives` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Derivatives Function Declaration Code

Code to be added to the declaration section of the generated `SystemDerivatives` function.

System Derivatives Function Execution Code

Code to be added to the execution section of the generated `SystemDerivatives` function.

System Derivatives Function Exit Code

Code to be added to the exit section of the generated `SystemDerivatives` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

See Also

`Model Header`, `Model Source`, `System Disable`, `System Enable`, `System Initialize`, `System Outputs`, `System Start`, `System Terminate`, `System Update`
“Insert Code into Model Custom Code Blocks”

Introduced in R2006a

System Disable

Specify custom system disable code

Library

Custom Code

Description

The System Disable block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemDisable` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Disable Function Declaration Code

Code to be added to the declaration section of the generated `SystemDisable` function.

System Disable Function Execution Code

Code to be added to the execution section of the generated `SystemDisable` function.

System Disable Function Exit Code

Code to be added to the exit section of the generated `SystemDisable` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

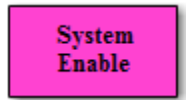
See Also

`Model Header`, `Model Source`, `System Derivatives`, `System Enable`, `System Initialize`, `System Outputs`, `System Start`, `System Terminate`, `System Update`
“Insert Code into Model Custom Code Blocks”

Introduced in R2006a

System Enable

Specify custom system enable code



Library

Custom Code

Description

The System Enable block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemEnable` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Enable Function Declaration Code

Code to be added to the declaration section of the generated `SystemEnable` function.

System Enable Function Execution Code

Code to be added to the execution section of the generated `SystemEnable` function.

System Enable Function Exit Code

Code to be added to the exit section of the generated `SystemEnable` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

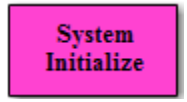
See Also

`Model Header`, `Model Source`, `System Derivatives`, `System Disable`, `System Initialize`, `System Outputs`, `System Start`, `System Terminate`, `System Update`
"Insert Code into Model Custom Code Blocks"

Introduced in R2006a

System Initialize

Specify custom system initialization code



Library

Custom Code

Description

The System Initialize block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemInitialize` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Initialize Function Declaration Code

Code to be added to the declaration section of the generated `SystemInitialize` function.

System Initialize Function Execution Code

Code to be added to the execution section of the generated `SystemInitialize` function.

System Initialize Function Exit Code

Code to be added to the exit section of the generated `SystemInitialize` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

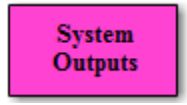
See Also

`Model Header`, `Model Source`, `System Derivatives`, `System Disable`, `System Enable`, `System Outputs`, `System Start`, `System Terminate`, `System Update`
“Insert Code into Model Custom Code Blocks”

Introduced in R2006a

System Outputs

Specify custom system outputs code



Library

Custom Code

Description

The System Outputs block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemOutputs` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Outputs Function Declaration Code

Code to be added to the declaration section of the generated `SystemOutputs` function.

System Outputs Function Execution Code

Code to be added to the execution section of the generated `SystemOutputs` function.

System Outputs Function Exit Code

Code to be added to the exit section of the generated `SystemOutputs` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

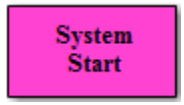
See Also

`Model Header`, `Model Source`, `System Derivatives`, `System Disable`, `System Enable`, `System Initialize`, `System Start`, `System Terminate`, `System Update`
“Insert Code into Model Custom Code Blocks”

Introduced in R2006a

System Start

Specify custom system startup code



Library

Custom Code

Description

The System Start block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemStart` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Start Function Declaration Code

Code to be added to the declaration section of the generated `SystemStart` function.

System Start Function Execution Code

Code to be added to the execution section of the generated `SystemStart` function.

System Start Function Exit Code

Code to be added to the exit section of the generated `SystemStart` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

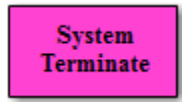
See Also

`Model Header`, `Model Source`, `System Derivatives`, `System Disable`, `System Enable`, `System Initialize`, `System Outputs`, `System Terminate`, `System Update`
"Insert Code into Model Custom Code Blocks"

Introduced in R2006a

System Terminate

Specify custom system termination code



Library

Custom Code

Description

The System Terminate block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemTerminate` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Terminate Function Declaration Code

Code to be added to the declaration section of the generated `SystemTerminate` function.

System Terminate Function Execution Code

Code to be added to the execution section of the generated `SystemTerminate` function.

System Terminate Function Exit Code

Code to be added to the exit section of the generated `SystemTerminate` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

See Also

`Model Header`, `Model Source`, `System Derivatives`, `System Disable`, `System Enable`, `System Initialize`, `System Outputs`, `System Start`, `System Update`
“Insert Code into Model Custom Code Blocks”

Introduced in R2006a

System Update

Specify custom system update code

Library

Custom Code

Description

The System Update block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemUpdate` function that the code generator creates for the model or subsystem that contains the block.

Note: If you include this block in a referenced model (model referenced by a Model block), the build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Update Function Declaration Code

Code to be added to the declaration section of the generated `SystemUpdate` function.

System Update Function Execution Code

Code to be added to the execution section of the generated `SystemUpdate` function.

System Update Function Exit Code

Code to be added to the exit section of the generated `SystemUpdate` function.

Example

See “Embed Custom Code Directly Into MdlStart Function”.

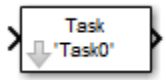
See Also

`Model Header`, `Model Source`, `System Derivatives`, `System Disable`, `System Enable`, `System Initialize`, `System Outputs`, `System Start`, `System Terminate`
"Insert Code into Model Custom Code Blocks"

Introduced in R2006a

Task Sync

Spawn an example RTOS (VxWorks) task to run code of downstream function-call subsystem or Stateflow chart



Library

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 might connect the Task Sync block to the output port of a Stateflow diagram that has an event, `Output to Simulink`, configured as a function call.

The Task Sync block performs the following functions:

- Uses the 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, using `semTake`. The first call to `semTake` specifies NO_WAIT. This allows the task to determine whether a second `semGive` has occurred prior to the completion of the function-call subsystem or chart. This would indicate that the interrupt rate is too fast or the task priority is too low.
- Generates synchronization code (for example, `semGive()`). This code allows the spawned task to run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. This is accomplished

through the connection between the Async Interrupt and Task Sync blocks, which triggers execution of the Task Sync block within an ISR.

- Supplies absolute time if blocks in the downstream algorithmic code require it. The time is supplied either by the timer maintained by the Async Interrupt block, or by an independent timer maintained by the task associated with the Task Sync block.

When you design your application, consider when timer and signal input values should be taken for the downstream function-call subsystem that is connected to the Task Sync block. By default, the time and input data are read when 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 is driven by an Async Interrupt block, the time and input data are read when the interrupt occurs (that is, within the ISR). For this case, data is synchronized with the caller of the Task Sync block.

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.

Parameters

Task name (10 characters or less)

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)

The RTOS task priority to be 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 to which the task's stack 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. You should determine the size by examining the generated code for the task (and functions that are called from the generated code).

Synchronize the data transfer of this task with the caller task

If not checked (the default),

- The block maintains a timer that provides absolute time values required by the computations of downstream blocks. The timer is independent of the timer maintained by the Async Interrupt block that calls the Task Sync block.
- A **Timer resolution** option appears.
- The **Timer size** option specifies the word size of the time counter.

If checked,

- The block does not maintain an independent timer, and does not display the **Timer resolution** field.
- Downstream blocks that require timers use the timer maintained by the Async Interrupt block that calls the Task Sync block (see "Timers in Asynchronous Tasks"). The timer value is read at the time the asynchronous interrupt is serviced, and data transfers to blocks called by the Task Sync block and execute within the task associated with the Async Interrupt block. Therefore, data transfers are synchronized with the caller.

Timer resolution (seconds)

The resolution of the block's timer in seconds. This option appears only if **Synchronize the data transfer of this task with the caller task** is not checked. By default, the block gets the timer value by calling the `tickGet` function in the RTOS (VxWorks). The default resolution is 1/60 second.

Timer size

The number of bits to be used to store the clock tick for a hardware timer. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the 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. However, when **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to

a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, it uses a second 32-bit integer to address overflows.

For more information, see “Control Memory Allocation for Time Counters”. See also “Timers in Asynchronous Tasks”.

Inputs and Outputs

Input

A call from an Async Interrupt block.

Output

A call to a function-call subsystem.

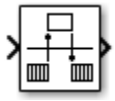
See Also

`Async Interrupt`
“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

4

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. These parameters require a Simulink Coder license. Additional parameters available with an ERT-based target require an Embedded Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Commonly Used** tab, on the **Code Generation** pane or on the **All Parameters** tab in the **Code Generation** category.

Parameter	Description
"System target file" on page 4-5	Specify which target file configuration will be used.
"Browse" on page 4-7	Browse file configuration options.
"Language" on page 4-8	Specify C or C++ code generation.
"Description" on page 4-10	A description of the target file.
"Toolchain" on page 4-11	Specify the toolchain to use when building an executable or library.
"Build configuration" on page 4-13	Specify compiler optimization or debug settings for toolchain.
"Tool/Options" on page 4-16	Display or customize build configuration settings.
"Compiler optimization level" on page 4-18	Control compiler optimizations for building generated code.
"Custom compiler optimization flags" on page 4-20	Specify custom compiler optimization flags.
"Generate makefile" on page 4-22	Enable generation of a makefile based on a template makefile.
"Make command" on page 4-24	Specify a make command and optionally append makefile options.
"Template makefile" on page 4-26	Specify the template makefile from which to generate the makefile.
"Select objective" on page 4-28	Select code generation objectives to use with the Code Generation Advisor.

Parameter	Description
"Prioritized objectives" on page 4-30	List objectives that you specify by clicking the **Set Objectives** button.
"Set Objectives" on page 4-31	Open Configuration Set Objectives dialog box.
"Set Objectives — Code Generation Advisor Dialog Box" on page 4-32	Select and prioritize code generation objectives.
"Check Model" on page 4-35	Check whether the model meets code generation objectives.
"Check model before generating code" on page 4-36	Choose whether to run Code Generation Advisor checks before generating code.
"Generate code only" on page 4-38	Specify code generation versus an executable build.
"Package code and artifacts" on page 4-40	Specify whether to automatically package generated code and artifacts for relocation.
"Zip file name" on page 4-42	Specify the name of the `.zip` file in which to package generated code and artifacts for relocation.

The Configuration Parameters dialog box also includes other code generation:

- on page 5-2
- on page 6-2
- on page 7-2
- on page 8-2
- on page 9-2

More About

- "Configuration Parameter Basics"

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.

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".
- To configure your model for rapid simulation, select `rsim.tlc`.
- To configure your model for Simulink Real-Time™, select `slrt.tlc` or `slrtert.tlc`.

Command-Line Information

Parameter: `SystemTargetFile`
Type: 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)

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Available System Target Files”

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 **on page 4-5**.

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` or `slrtert.tlc`.

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Select a System Target File”
- “Available System Target Files”

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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Change Programming Language”
- “Select and Configure Compiler or IDE”
- “Control Generation of Function Prototypes”
- “Control Generation of C++ Class Interfaces”

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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Browse” on page 4-7

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 "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 **All Parameters** > **Toolchain** > **Validate** 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 in the validation process. To fix a failure, 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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Toolchain Configuration”
- “Adding a Custom Toolchain”

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 "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 4-16.

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 4-16.

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 "Files and Folders Created During Build Process".

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 4-16.

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

Related Examples

- "Model Configuration Parameters: Code Generation" on page 4-2

- “Toolchain Configuration”
- “Adding a Custom Toolchain”

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 "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: `''`

Related Examples

- "Model Configuration Parameters: Code Generation" on page 4-2

- “Toolchain Configuration”
- “Adding a Custom Toolchain”

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 "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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Custom compiler optimization flags” on page 4-20
- “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 "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 4-18.

Related Examples

- "Model Configuration Parameters: Code Generation" on page 4-2

- “Compiler optimization level” on page 4-18
- “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 "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

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 "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

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 "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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Template Makefiles and Make Options”
- “Available System Target Files”

Select objective

Description

Select code generation objectives to use 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.

`Debugging`

Specifies debugging objective. Optimize code generation settings for debugging the code generation build process using the Code Generation Advisor.

`Execution efficiency`

Specifies execution efficiency objective. Optimize code generation settings to achieve fast execution time using the Code Generation Advisor.

Tips

For more objectives, specify an ERT-based target.

Dependency

These parameters appear only for GRT-based targets.

Command-Line Information

Parameter: `'ObjectivePriorities'`
Type: cell array of character vectors
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

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives Using Code Generation Advisor”
- “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')`

Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives Using Code Generation Advisor”
- “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.

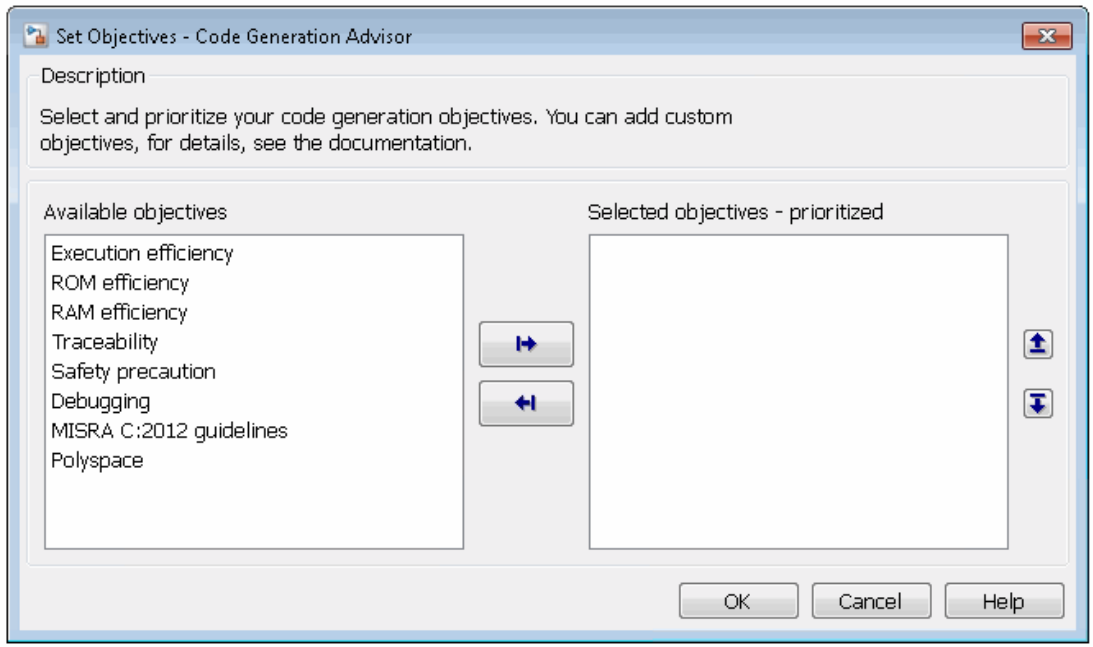
Related Examples

- “Model Configuration Parameters: Code Generation” on page 4-2
- “Configure Model for Code Generation Objectives Using Code Generation Advisor”
- “Application Objectives Using Code Generation Advisor”

Set Objectives — Code Generation Advisor Dialog Box

Description

Select and prioritize code generation objectives to use with the Code Generation Advisor.



Category: Code Generation

Settings

1. From the **Available objectives** list, select objectives.
2. Click the select button (arrow pointing right) to move the objectives that you selected into the **Selected objectives - prioritized** list.
3. Click the higher priority (up arrow) and lower priority (down arrow) buttons to prioritize the objectives.

Objectives

List of available objectives.
`Execution efficiency` — Configure code generation settings to achieve fast execution time.
`ROM efficiency` — Configure code generation settings to reduce ROM usage.
`RAM efficiency` — Configure code generation settings to reduce RAM usage.
`Traceability` — Configure code generation settings to provide mapping between model elements and code.
`Safety precaution` — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
`Debugging` — Configure code generation settings to debug the code generation build process.
`MISRA C: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; combination of the available values
Value: `{''} | {'Execution efficiency'} | {'ROM efficiency'} | {'RAM efficiency'} | {'Traceability'} | {'Safety precaution'} | {'Debugging'} | {'MISRA C:2012 guidelines'} | {'Polyspace'}`
Default: `{''}`

Related Examples

- "Model Configuration Parameters: Code Generation" on page 4-2
- "Configure Model for Code Generation Objectives Using Code Generation Advisor"
- "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.

Related Examples

- "Model Configuration Parameters: Code Generation" on page 4-2
- "Configure Model for Code Generation Objectives Using Code Generation Advisor"
- "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)`

Related Examples

- "Model Configuration Parameters: Code Generation" on page 4-2
- "Configure Model for Code Generation Objectives Using Code Generation Advisor"
- "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

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
Efficiency	No impact

Application	Setting
Safety precaution	No impact

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
Safety precaution	No impact

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 **Commonly Used** tab on the **Code Generation > Report** pane, or on the **All Parameters** tab in the **Code Generation > Report** category.

Parameter	Description
"Create code generation report" on page 5-4	Document generated code in an HTML report.
"Open report automatically" on page 5-7	Specify whether to display code generation reports automatically.
"Generate model Web view" on page 5-9	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 5-11	Include static code metrics report in the code generation report.

See Also

"Code-to-model" on page 10-13 | "Model-to-code" on page 10-15 | "Eliminated / virtual blocks" on page 10-18 | "Traceable Stateflow objects" on page 10-22 | "Traceable MATLAB functions" on page 10-24

More About

- "Report Generation"
- "Configuration Parameter Basics"

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.

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"

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 5-7
- "Code-to-model" on page 10-13 (ERT target)

This parameter enables

- "Model-to-code" on page 10-15 (ERT target)
- "Eliminated / virtual blocks" on page 10-18 (ERT target)
- "Traceable Simulink blocks" on page 10-20 (ERT target)
- "Traceable Stateflow objects" on page 10-22 (ERT target)
- "Traceable MATLAB functions" on page 10-24 (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

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions”
- “Configure Code Coverage with Third-Party Tools”

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

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Reports for Code Generation”
- “HTML Code Generation Report Extensions”

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 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"

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

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “Static Code Metrics”

6

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 **Commonly Used** tab, on the **Code Generation > Comments** pane or on the **All Parameters** tab in the **Code Generation > Comments** category.

Parameter	Description
"Include comments" on page 6-5	Specify which comments are in generated files.
"Simulink block / Stateflow object comments" on page 6-7	Specify whether to insert Simulink block and Stateflow object comments.
"MATLAB source code as comments" on page 6-9	Specify whether to insert MATLAB source code as comments.
"Show eliminated blocks" on page 6-11	Specify whether to insert eliminated block's comments.
"Verbose comments for SimulinkGlobal storage class" on page 6-13	Reduce code size or improve code traceability by controlling the generation of comments.
"Operator annotations" on page 6-15	Specify whether to include operator annotations for Polyspace in the generated code as comments.
"Simulink block descriptions" on page 6-17	Specify whether to insert descriptions of blocks into generated code as comments.
"Simulink data object descriptions" on page 6-19	Specify whether to insert descriptions of Simulink data objects into generated code as comments.
"Custom comments (MPT objects only)" on page 6-21	Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.

Parameter	Description
"Custom comments function" on page 6-23	Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects.
"Stateflow object descriptions" on page 6-25	Specify whether to insert descriptions of Stateflow objects into generated code as comments.
"Requirements in block comments" on page 6-27	Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.
"MATLAB function help text" on page 6-29	Specify whether to include MATLAB function help text in the function banner.

More About

- "Code Appearance"
- "Configuration Parameter Basics"

Code Generation: Comments Tab Overview

Control the comments that the code generator creates and inserts into the generated code.

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 / Stateflow object comments**
- **MATLAB source code as comments**
- **Show eliminated blocks**
- **Verbose comments for SimulinkGlobal storage class**

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

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2

Simulink block / Stateflow object comments

Description

Specify whether to insert Simulink block and Stateflow object comments.

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts automatically generated comments that describe a block's code and objects. The comments precede that code in the generated file.

 Off

Suppresses comments.

Dependency

This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: `SimulinkBlockComments`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact

Application	Setting
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2

MATLAB source code as comments

Description

Specify whether to insert MATLAB source code as comments.

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts MATLAB source code as comments in the generated code. The comments precede the associated generated code.

Includes the function signature in the function banner.

 Off

Suppresses comments and does not include the function signature in the function banner.

Dependency

This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: `MATLABSourceComments`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

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

This parameter is enabled by **Include comments**.

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

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2

Verbose comments for SimulinkGlobal 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 SimulinkGlobal to the object or variable.
- You apply the storage class Auto to the object or variable and set the model configuration parameter **Default parameter behavior** to Tunable.

For more information about parameter representation in the generated code, see "Block Parameter Representation in the Generated Code".

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 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

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Block Parameter Representation in the Generated Code”

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.
- This parameter is enabled by **Include comments**.

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

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Annotate Code for Justifying Polyspace Checks”

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 / Stateflow object 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	No impact
Safety precaution	No impact

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	No impact
Safety precaution	No impact

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	No impact
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Add Custom Comments for Variables in the Generated Code”

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.
- This parameter is enabled by **Custom comments (MPT objects only)**.

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	No impact
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “Add Custom Comments for Variables in the Generated Code”

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	No impact
Safety precaution	No impact

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 Verification and Validation™ 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	No impact
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Code Generation Comments” on page 6-2
- “How Requirements Information Is Included in Generated Code”

MATLAB function help text

Description

Specify whether to include MATLAB function help text in the function banner.

Category: Code Generation > Comments

Settings

Default: On

 On

Inserts MATLAB function help text in the function banner.

 Off

Inserts MATLAB function help text in the body of the function.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Include comments**.

Command-Line Information

Parameter: `MATLABFcnDesc`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

Related Examples

- "Model Configuration Parameters: Code Generation Comments" on page 6-2
- "Including MATLAB Function Help Text in the Function Banner"

7

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 **Commonly Used** tab on the **Code Generation > Symbols** pane, or on the **All Parameters** tab in the **Code Generation > Symbols** category.

Parameter	Description
"Global variables" on page 7-5	Customize generated global variable identifiers.
"Global types" on page 7-8	Customize generated global type identifiers.
"Field name of global types" on page 7-11	Customize generated field names of global types.
"Subsystem methods" on page 7-13	Customize generated function names for reusable subsystems.
"Subsystem method arguments" on page 7-16	Customize generated function argument names for reusable subsystems.
"Local temporary variables" on page 7-18	Customize generated local temporary variable identifiers.
"Local block output variables" on page 7-21	Customize generated local block output variable identifiers.
"Constant macros" on page 7-23	Customize generated constant macro identifiers.
"Shared utilities" on page 7-26	Customize shared utility identifiers.
"Minimum mangle length" on page 7-29	Specify the minimum number of characters for generating name-mangling text to help avoid name collisions.
"Maximum identifier length" on page 7-31	Specify maximum number of characters in generated function, type definition, variable names.

Parameter	Description
"System-generated identifiers" on page 7-33	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 7-38	Control expression of scalar inlined parameter values in the generated code.
"Signal naming" on page 7-41	Specify rules for naming signals in generated code.
"M-function" on page 7-43	
"Parameter naming" on page 7-45	Specify rule for naming parameters in generated code.
"M-function" on page 7-47	
"#define naming" on page 7-49	Specify rule for naming `#define` parameters (defined with storage class `Define (Custom)`) in generated code.
"M-function" on page 7-51	
"Use the same reserved names as Simulation Target" on page 7-53	Specify whether to use the same reserved names as those specified in the **Simulation Target** pane.
"Reserved names" on page 7-55	Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

More About

- "Code Appearance"
- "Configuration Parameter Basics"

Code Generation: Symbols Tab Overview

Select the automatically generated identifier naming rules.

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"

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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.

- 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.
- This parameter requires an Embedded Coder license 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

Application	Setting
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Related Examples

- "Model Configuration Parameters: Code Generation Symbols" on page 7-2
- "Identifier Format Control"
- "Control Name Mangling in Generated Identifiers"
- "Avoid Identifier Name Collisions with Referenced Models"
- "Identifier Format Control Parameters Limitations"

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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.

- 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.
- This parameter requires an Embedded Coder license 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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.
- The **Maximum identifier length** setting does not apply to type definitions.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `CustomSymbolStrField`
Type: 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Identifier Format Control Parameters Limitations”

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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.
- 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.
- This parameter requires an Embedded Coder license 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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.

Dependencies

This parameter:

- Appears only for ERT-based targets.
- Requires an Embedded Coder license 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Identifier Format Control Parameters Limitations”

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `CustomSymbolStrTmpVar`
Type: 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control”

- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `CustomSymbolStrBlkIO`
Type: 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Identifier Format Control Parameters Limitations”

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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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.

- 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.
- This parameter requires an Embedded Coder license 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

Application	Setting
Efficiency	No impact
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Identifier Format Control”
- “Control Name Mangling in Generated Identifiers”
- “Avoid Identifier Name Collisions with Referenced Models”
- “Identifier Format Control Parameters Limitations”

Shared utilities

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`. 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. This parameter is on the **All Parameters** tab of the Configuration Parameters dialog box.

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.

- 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 `$N` until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.
- Descriptive text helps make the identifier name more accessible.
- For versions prior to R2016a, the **Shared utilities** parameter does not support the `$R` token. For a model, if the **Shared utilities** parameter includes a `$R` token, and you export the model to a version prior to R2016a, the **Shared utilities** parameter defaults to `$N$C`.

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `CustomSymbolStrUtil`
Type: character vector
Value: valid combination of tokens
Default: `$N$C`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Use default
Efficiency	No impact
Safety precaution	No recommendation

Related Examples

- "Model Configuration Parameters: Code Generation Symbols" on page 7-2
- "Identifier Format Control"
- "Exceptions to Identifier Formatting Conventions"

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.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `MangleLength`
Type: integer
Value: value between 1 and 15

Default: 1

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	1
Efficiency	No impact
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Control Name Mangling in Generated Identifiers”
- “Maintain Traceability for Generated Identifiers”

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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Construction of Generated Identifiers”
- “Identifier Name Collisions and Mangling”
- “Identifier Format Control”

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 7-5 is `$N$R$M`, the block state identifier is `sym_DWork`.
- "Global types" on page 7-8 is `$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 7-5 is `$N$R$M`, the block state identifier is `sym_DW`.
- "Global types" on page 7-8 is `$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

Classic	Shortened	Data Representation	Description
`ExternalInputSizes`	`ExtUSize`	Type	Size of block input data for the root system (used when inputs are variable dimensions)
`ExternalOutputs`	`ExtY`	Type	Block output data for the root system
`ExternalOutputSizes`	`ExtYSize`	Type	Size of block output data for the root system
`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
`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

Classic	Shortened	Data Representation	Description
`StateDisabled`, `Xdis`	`XDis`	Type, Global Variable	Status of an enabled subsystem
`StateDerivatives`, `Xdot`	`XDot`	Type, Global Variable	Derivatives of continuous states at each time step
`ZCSignalValues`, `ZCSignalValue`	`ZCV`	Type, Global Variable	Zero-crossing signals
`DefaultParameters`	`DefaultP`	Global Variable	Default parameters in the system
`GlobalTID`	`GlobalTID`	Global Variable	Used for sample time for states in referenced models
`InvariantSignals`	`Invariant`	Global Variable	Invariant signals
`NSTAGES`	`NSTAGES`	Global Variable	Solver macro
`Object`	`Obj`	Global Variable	Used by ERT C++ code generation to refer to referenced model objects
`TimingBridge`	`TimingBrdg`	Global Variable	Timing information stored in different data structures

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

Classic	Shortened	Data Representation	Description
`rtP, P`	`P`	Type, Global Variable	Parameters for the system
`rtRTM, RTM`	`RTM`	Type, Global Variable	RT_Model structure
`rtX, X`	`X`	Type, Global Variable	Continuous states in model reference
`rtXdis, Xdis`	`XDis`	Type, Global Variable	Status of an enabled subsystem
`rtXdot, Xdot`	`XDot`	Type, Global Variable	Derivatives of the S-function's continuous states at each time step
`rtZCE, ZCE`	`ZCE`	Type, Global Variable	Zero-crossing enabled
`rtZCSV, ZCSV`	`ZCV`	Type, Global Variable	Zero-crossing signal values

Dependencies

- This parameter appears only for ERT-based targets.
- When generating code, this parameter requires an Embedded Coder license.

Command-Line Information

Parameter: `InternalIdentifier`
Type: character vector
Value: `Classic | Shortened`
Default: `Shortened`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

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"
- "Default Data Structures in the Generated Code"
- "Customize Generated Identifier Naming Rules"
- "Identifier Format Control"

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 > Signals and Parameters > 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.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `InlinedPrmAccess`
Type: character vector
Value: `Literals` | `Macros`
Default: `Literals`

Recommended Settings

Application	Setting
Debugging	No impact

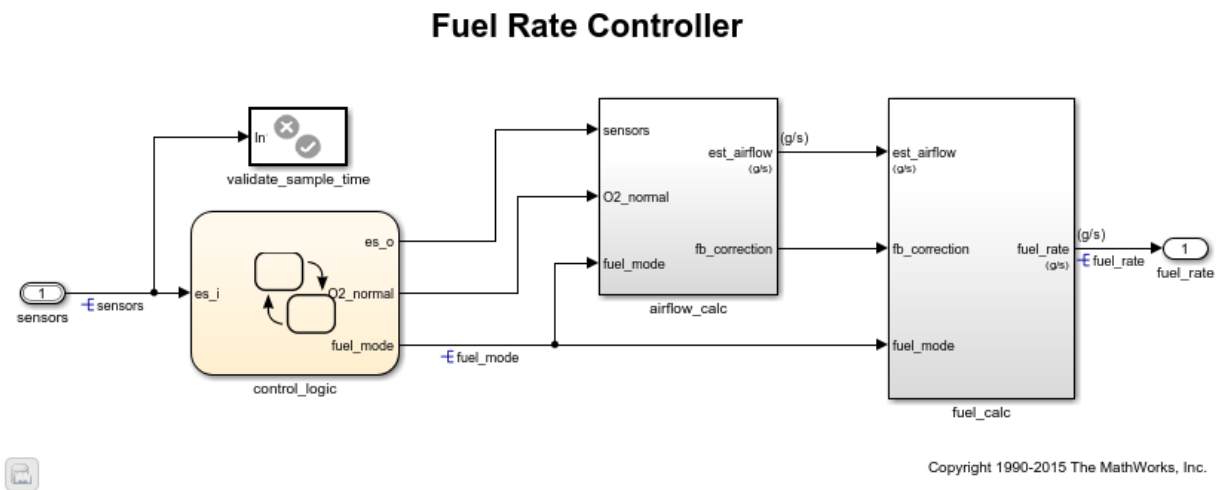
Application	Setting
Traceability	`Macros`
Efficiency	No impact
Safety precaution	No impact

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.

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2

Signal naming

Description

Specify rules for naming 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 `Simulink.Signal` data object, that value overrides the specification of the **Signal naming** parameter.

Limitation

This parameter does not impact signal names that are specified by an embedded signal object created using the **Code Generation** tab of a **Signal Properties** dialog box. See "Programmatically Apply Custom Storage Classes Directly to Signals, States, and Outport Blocks Using Embedded Signal Objects" for information about embedded signal 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

Related Examples

- "Model Configuration Parameters: Code Generation Symbols" on page 7-2
- "Apply Naming Rules to Simulink Data Objects"
- "Programming Scripts and Functions"

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.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Signal naming**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `SignalNamingFcn`
Type: character vector
Value: MATLAB language file
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule Using a Function”
- “Programming Scripts and Functions”

Parameter naming

Description

Specify rule for naming parameters in generated code.

This parameter does not affect `Simulink.LookupTable` or `Simulink.Breakpoint` objects.

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.

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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Apply Naming Rules to Simulink Data Objects”
- “Programming Scripts and Functions”

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.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Parameter naming**.
- This parameter 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule Using a Function”
- “Programming Scripts and Functions”

#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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule for Storage Class Define”
- “Programming Scripts and Functions”

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.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **#define naming**.
- This parameter 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

Related Examples

- “Model Configuration Parameters: Code Generation Symbols” on page 7-2
- “Specify Naming Rule Using a Function”
- “Programming Scripts and Functions”

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

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
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

Related Examples

- “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 **Commonly Used** tab, on the **Code Generation > Custom Code** pane or on the **All Parameters** tab in the **Code Generation > Custom Code** category.

Parameter	Description
"Use the same custom code settings as Simulation Target" on page 8-4	Specify whether to use the same custom code settings as those in the **Simulation Target > Custom Code** pane.
"Source file" on page 8-6	Specify custom code to include near the top of the generated model source file.
"Header file" on page 8-7	Specify custom code to include near the top of the generated model header file.
"Initialize function" on page 8-9	Specify custom code to include in the generated model initialize function.
"Terminate function" on page 8-10	Specify custom code to include in the generated model terminate function.
"Include directories" on page 8-12	Specify a list of include folders to add to the include path.
"Source files" on page 8-14	Specify a list of additional source files to compile and link with the generated code.
"Libraries" on page 8-16	Specify a list of additional libraries to link with the generated code.
"Defines" on page 8-18	Specify preprocessor macro definitions to be added to the compiler command line.

More About

- "Configuration Parameter Basics"

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**.

Related Examples

- "Model Configuration Parameters: Code Generation Custom Code" on page 8-2
- "Configure Model for External Code Integration"

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

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Configure Model for External Code Integration”

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

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Configure Model for External Code Integration”

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

Related Examples

- "Model Configuration Parameters: Code Generation Custom Code" on page 8-2
- "Configure Model for External Code Integration"

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

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Configure Model for External Code Integration”

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 on the **All Parameters** tab.

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

Related Examples

- "Model Configuration Parameters: Code Generation Custom Code" on page 8-2
- "Configure Model for External Code Integration"

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

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2
- “Configure Model for External Code Integration”

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
Safety precaution	No impact

Related Examples

- "Model Configuration Parameters: Code Generation Custom Code" on page 8-2
- "Configure Model for External Code Integration"

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
Safety precaution	No impact

Related Examples

- "Model Configuration Parameters: Code Generation Custom Code" on page 8-2
- "Configure Model for External Code Integration"

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

Related Examples

- “Model Configuration Parameters: Code Generation Custom Code” on page 8-2

- “Configure Model for External Code Integration”

9

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 **Commonly Used** tab, on the **Code Generation > Interface** pane.

Parameter	Description
"Code replacement library" on page 9-6	Specify a code replacement library the code generator uses when producing code for a model.
"Shared code placement" on page 9-9	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 9-11	Specify whether to generate floating-point data and operations.
"Support: non-finite numbers" on page 9-13	Specify whether to generate non-finite data and operations on non-finite data.
"Support: complex numbers" on page 9-15	Specify whether to generate complex data and operations.
"Support: absolute time" on page 9-17	Specify whether to generate and maintain integer counters for absolute and elapsed time values.
"Support: continuous time" on page 9-19	Specify whether to generate code for blocks that use continuous time.
"Support: variable-size signals" on page 9-22	Specify whether to generate code for models that use variable-size signals.
"Code interface packaging" on page 9-24	Select the packaging for the generated C or C++ code interface.
"Multi-instance code error diagnostic" on page 9-28	Select the severity level for diagnostics displayed when a model violates

Parameter	Description
	requirements for generating multi-instance code.
"Pass root-level I/O as" on page 9-30	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 9-32	Specify whether to log or monitor error status.
"Configure Model Functions" on page 9-34	Specify whether the code generator uses default *model*_initialize and *model*_step function prototypes or model-specific C prototypes.
"Parameter visibility" on page 9-35	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 9-37	Specify whether to generate access methods for block parameters for the C++ model class.
"External I/O access" on page 9-39	Specify whether to generate access methods for root-level I/O signals for the C ++ model class.
"Configure C++ Class Interface" on page 9-41	Customize the C++ class interface for your model code.
"Generate C API for: signals" on page 9-42	Generate C API data interface code with a signals structure.
"Generate C API for: parameters" on page 9-44	Generate C API data interface code with parameter tuning structures.
"Generate C API for: states" on page 9-46	Generate C API data interface code with a states structure.
"Generate C API for: root-level I/O" on page 9-48	Generate C API data interface code with a root-level I/O structure.
"ASAP2 interface" on page 9-50	Generate code for the ASAP2 data interface.

Parameter	Description
“External mode” on page 9-52	Generate code for the external mode data interface.
“Transport layer” on page 9-54	Specify the transport protocol for communications.
“MEX-file arguments” on page 9-56	Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.
“Static memory allocation” on page 9-58	Control memory buffer for external mode communication.
“Static memory buffer size” on page 9-60	Specify the memory buffer size for external mode communication.

More About

- “Configuration Parameter Basics”

Code Generation: Interface Tab Overview

Select the target software environment, output variable name modifier, and data exchange interface.

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Target 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.

`GNU C99 extensions`

Generates calls to the GNU® gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

`AUTOSAR 4.0`

Produces code that more closely aligns with the AUTOSAR standard. Requires an Embedded Coder license.

`Intel IPP for x86-64 (Windows)`

Generates calls to the Intel® Performance Primitives (IPP) library for the x86-64 Windows platform.

`Intel IPP/SSE for x86-64 (Windows)`

Generates calls to the IPP and Streaming SIMD Extensions (SSE) libraries for the x86-64 Windows platform.

`Intel IPP for x86-64 (Windows using MinGW compiler)`

Generates calls to the IPP library for the x86-64 Windows platform and MinGW compiler.

`Intel IPP/SSE for x86-64 (Windows using MinGW compiler)`

Generates calls to the IPP and SSE libraries for the x86/Pentium Windows platform.

`Intel IPP for x86/Pentium (Windows)`

Intel IPP for x86/Pentium (Windows)—Generates calls to the IPP library for the x86/Pentium Windows platform.

`Intel IPP/SSE for x86/Pentium (Windows)`

Intel IPP for x86/Pentium (Windows)—Generates calls to the IPP and SSE libraries for the x86/Pentium Windows platform.

`Intel IPP for x86-64 (Linux)`

Generates calls to the IPP library for the x86-64 Linux® platform.

`Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)`

Generates calls to the GNU libraries for IPP and SSE, with GNU C99 extensions, for the x86-64 Linux platform.

- Additional values might be listed for licensed target products and for embedded and desktop targets. If you have created and registered code replacement libraries using the Embedded Coder product, additional values are listed.
- The software filters the list of **Code replacement library** values based on compatibility with **Language**, **Standard math library**, and **Device vendor** values you select for your model.

Tips

- If you specify `Shared location` for the **Code Generation** > **Interface** > **Shared code placement** parameter or you generate code for models in a model reference hierarchy,
  - Models that are sharing the location or are in the model hierarchy must specify the same code replacement library (same name, tables, and table entries).
  - 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 "Shared Utility 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.

Tip

Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Command-Line Information

Parameter: `CodeReplacementLibrary`
Type: 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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Target Environment Configuration”
- “What Is Code Replacement Customization?”
- “Develop a Code Replacement Library”

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`

Operates as follows:

- When the model contains Model blocks or `ExistingSharedCode` is not empty, 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` (GRT) No impact (ERT)
Traceability	`Shared location` (GRT) No impact (ERT)

Application	Setting
Efficiency	No impact (execution, RAM) `Shared location` (ROM)
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Target Environment Configuration”
- “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

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**.
- This parameter 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, 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

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

Command-Line Information

Parameter: `SupportComplex`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	Off (for real only)
Safety precaution	No impact

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

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Use Discrete and Continuous Time”

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

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**.
- 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
- “Entry-Point Functions and Scheduling”
- “Generate Reentrant Code from Top-Level Models”
- “Use GRT with Reusable Function Packaging to Combine Models”
- “Generate Reentrant Code from Top-Level Models”

- “Generate C++ Class Interface to Model or Subsystem Code”
- “Control Generation of C++ Class Interfaces”
- “Code Generation of Subsystems”
- “Code Reuse Limitations for Subsystems”
- “Determine Why Subsystem Code Is Not Reused”
- “S-Functions That Support Code Reuse”
- “Static Main Program Module”
- “Control Generation of Function Prototypes”
- “Generate Modular Function Code”
- “Export Function-Call Subsystems”

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

Related Examples

- "Model Configuration Parameters: Code Generation Interface" on page 9-2
- "Entry-Point Functions and Scheduling"
- "Generate Reentrant Code from Top-Level Models"
- "Generate C++ Class Interface to Model or Subsystem Code"
- "Code Generation of Subsystems"
- "Code Reuse Limitations for Subsystems"
- "Determine Why Subsystem Code Is Not Reused"
- "Generate Modular Function Code"

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
- “Entry-Point Functions and Scheduling”
- “Generate Reentrant Code from Top-Level Models”
- “Code Generation of Subsystems”
- “Generate Modular Function Code”

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Use the Real-Time Model Data Structure”

Configure Model Functions

Description

Open the Model Interface dialog box. In this dialog box, you can specify whether the code generator uses default *model*_initialize and *model*_step function prototypes or model-specific C prototypes. Based on your selection, you can preview and modify the function prototypes.

Category: Code Generation > Interface

Dependencies

- This button appears only for ERT-based targets with **Code interface packaging** set to a value other than `C++ class`.
- This button requires an Embedded Coder license when generating code.
- This button is active only if your model uses an attached configuration set. If your model uses a referenced configuration set, the button is greyed out. If you want to configure a model-specific step function prototype for a referenced configuration set, use the MATLAB function prototype control functions described in "Configure Function Prototypes Programmatically".

See Also

`model_initialize` | `model_step`

Related Examples

- "Model Configuration Parameters: Code Generation Interface" on page 9-2
- "Control Generation of Function Prototypes"
- "Launch the Model Interface Dialog Boxes"

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options”

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`

Application	Setting
Traceability	`Inlined method`
Efficiency	`Inlined method`
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options”

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* affect 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" and "Passing I/O Arguments".

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure Code Interface Options”

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".

See Also

`model_step`

Related Examples

- "Model Configuration Parameters: Code Generation Interface" on page 9-2
- "Control Generation of C++ Class Interfaces"
- "Configure Step Method for Your Model Class"

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

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

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

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

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

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2

- “Set Up and Use Host/Target Communication Channel”

Transport layer

Description

Specify the transport protocol for communications.

Category: Code Generation > Interface

Settings

Default: `tcpip`

`tcpip`

Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

`serial`

Applies a serial transport mechanism. The MEX-file name is `ext_serial_win32_comm`.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. The value is specified either in *`matlabroot`*`/toolbox/simulink/simulink/extmode_transports.m`, for targets provided by MathWorks®, or in an `sl_customization.m` file, for custom targets and/or custom transports.

Dependency

Selecting **External mode** enables this parameter.

Command-Line Information

Parameter: `ExtModeTransport`
Type: integer
Value: `0` for TCP/IP | `1` for serial
Default: `0`

Recommended Settings

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Target Interfacing”
- “Create a Transport Layer for External 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 TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target (for example, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

For a serial transport, `ext_serial_win32_comm` allows three optional arguments:

- Verbosity level (`0` for no information or `1` for detailed information)
- Serial port ID (`1` for `COM1`, and so on)
- Baud (selected from the set `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600`, `115200`, with a default baud of `57600`)

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Target Interfacing”
- “Choose Communication Protocol for Client and Server”

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**.

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

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure External Mode Options for Code Generation”

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

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Code Generation Interface” on page 9-2
- “Configure External Mode Options for Code Generation”

10

Simulink Coder Parameters: All Parameters Tab Only

Model Configuration Parameters: Advanced Parameters

Simulink Coder provides advanced configuration parameters that appear only on the **All Parameters** tab of the Configuration Parameters dialog box. These advanced parameters are in the Code Generation and Hardware Implementation categories. To view these parameters, on the **All Parameters** tab, search for `Advanced Parameters`.

In this section...

Configuration Parameters for Code Generation Advanced Parameters

Parameter	Description
“Classic call interface” on page 10-36	Specify whether to generate model function calls compatible with the main program module of the GRT target in models created before R2012a.
“Code-to-model” on page 10-13	Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.
“Combine signal/state structures” on page 10-47	Specify whether to combine global block signals and global state data into one data structure in the generated code
“Configure” on page 10-17	Open the **Model-to-code navigation** dialog box for specifying a build folder containing previously-generated model code to highlight.
“Custom LAPACK library callback” on page 10-73	Specify LAPACK library callback class for LAPACK calls in code generated from MATLAB code.

Parameter	Description
"Eliminated / virtual blocks" on page 10-18	Include summary of eliminated and virtual blocks in code generation report.
"Enable TLC assertion" on page 10-71	Produce the TLC stack trace.
"Generate destructor" on page 10-54	Specify whether to generate a destructor for the C++ model class.
"Ignore custom storage classes" on page 10-9	Specify whether to apply or ignore custom storage classes.
"Ignore test point signals" on page 10-11	Specify allocation of memory buffers for test points.
"Internal data access" on page 10-52	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 10-50	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.
"MAT-file logging" on page 10-56	Specify MAT-file logging.
"MAT-file variable name modifier" on page 10-59	Select the text to add to MAT-file variable names.
"Maximum word length" on page 10-34	Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.
"Model-to-code" on page 10-15	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
"Multiword type definitions" on page 10-32	Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.
"Profile TLC" on page 10-65	Profile the execution time of TLC files.
"Retain .rtw file" on page 10-63	Specify *`model`*`.rtw` file retention.
"Single output/update function" on page 10-42	Specify whether to generate the *`model`*`_step` function.
"Standard math library" on page 10-28	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
"Start TLC coverage when generating code" on page 10-69	Generate the TLC execution report.
"Start TLC debugger when generating code" on page 10-67	Specify use of the TLC debugger
"Summarize which blocks triggered code replacements" on page 10-26	Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.
"Support: non-inlined S-functions" on page 10-30	Specify whether to generate code for non-inlined S-functions.
"Terminate function required" on page 10-45	Specify whether to generate the *`model`*`_terminate` function.
"Traceable MATLAB functions" on page 10-24	Include summary of MATLAB functions in code generation report.
"Traceable Simulink blocks" on page 10-20	Include summary of Simulink blocks in code generation report.

Parameter	Description
"Traceable Stateflow objects" on page 10-22	Include summary of Stateflow objects in code generation report.
"Use dynamic memory allocation for model block instantiation" on page 10-40	Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ class interface.
"Use dynamic memory allocation for model initialization" on page 10-38	Control how the generated code allocates memory for model data.
"Use Simulink Coder Features" on page 10-75	Enable "Simulink Coder" features for models deployed to "Simulink Supported Hardware".
"Verbose build" on page 10-61	Display code generation progress.
"Custom token text"	Custom text to replace `$U` in the Symbols pane.
"Remove reset function"	Remove unreachable (dead-code) instances of the `reset` functions from the generated code for ERT-based systems that include model referencing hierarchies.
"Remove disable function"	Remove unreachable (dead-code) instances of the `disable` functions from the generated code for ERT-based systems that include model referencing hierarchies.

The following Code Generation > Advanced Parameters are infrequently used and have no other documentation.

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 on the **All Parameters** tab to control the compiler optimization level for building generated code.

Parameter	Description
	Default is `off` for custom targets and `on` for targets provided with the Simulink Coder and Embedded Coder products.
`ConcurrentExecutionCompliant`	Indicates whether the target supports concurrent execution.
`GenerateFullHeader`	Generate full header including time stamp.
`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.
`IsERTTarget`	Indicates whether or not the currently selected target is derived from the ERT target.
`MultiwordLength`	Maximum multiword length.
`ModelReferenceCompliant` character vector - `off`, **`on`**	Set in `SelectCallback` for a target to indicate whether the target supports model reference.
`ParMdlRefBuildCompliant`	Indicates if the model is configured for parallel builds when building a model that includes referenced models.
`PostCodeGenCommand` *`character vector`* - `''`	Add the specified post code generation command to the model build process.
`ProfileTLC` character vector - **`off`**, `on`	Profile the execution time of each TLC file used to generate code for this model in HTML format.
`RTWVerbose` character vector - `off`, **`on`**	Display messages indicating code generation stages and compiler output.
`RetainRTWFile` character vector - **`off`**, `on`	Retain the *`model`*`.rtw` file in the current build folder.

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:** To use this parameter with 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`.
`TLCAssert` character vector - **`off`**, `on`	Produce a TLC stack trace when the argument to the `assert` directives evaluates to `false`.
`TLCCoverage` character vector - **`off`**, `on`	Generate `.log` files containing the number of times each line of TLC code is executed during code generation.
`TLCDebug` character vector - **`off`**, `on`	Start the TLC debugger during code generation at the beginning of the TLC program. TLC breakpoint statements automatically invoke the TLC debugger regardless of this setting.
`TLCOptions` *`character vector`* - `''`	Specify additional TLC command line options.

Configuration Parameters for Hardware Implementation Advanced Parameters

The following model configuration parameters have no other documentation.

Parameter	Description
`TargetPreprocMaxBitsSint` int - **32**	Specify the maximum number of bits that the target C preprocessor can use for signed integer math.
`TargetPreprocMaxBitsUint` int - **32**	Specify the maximum number of bits that the target C preprocessor can use for unsigned integer math.
"Use Embedded Coder Features"	Enable "Embedded Coder" features for models deployed to "Simulink Supported Hardware".

Configuration Parameters for MathWorks Use Only

Parameter	Description
`Comment`	For MathWorks use only.
`ERTCustomFileBanners`	For MathWorks use only.
`ExtModeMexFile`	For MathWorks use only.
`ExtModeTesting`	For MathWorks use only.
`IncAutoGenComments`	For MathWorks use only.
`PreserveName`	For MathWorks use only.
`PreserveNameWithParent`	For MathWorks use only.
`SignalNamingFcn`	For MathWorks use only.
`TargetFcnLib`	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.

Ignore custom storage classes

Description

Specify whether to apply or ignore custom storage classes.

Category: Code Generation

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Custom Storage Classes”

Ignore test point signals

Description

Specify allocation of memory buffers for test points.

Category: Code Generation

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

Application	Setting
Traceability	No impact
Efficiency	On
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Signals with Test Points”
- “Test Points”
- “Signal Representation in Generated Code”

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 > Advanced Parameters

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**.
- You must select **Include comments** 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

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions”

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 > Advanced Parameters

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**.
- 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 10-18
    - "Traceable Simulink blocks" on page 10-20
    - "Traceable Stateflow objects" on page 10-22

- "Traceable MATLAB functions" on page 10-24

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

Related Examples

- "Model Configuration Parameters: Code Generation Report" on page 5-2
- "HTML Code Generation Report Extensions"

Configure

Description

Open the **Model-to-code navigation** dialog box. This dialog box provides a way for you to specify a build folder containing previously-generated model code to highlight. Applying your build folder selection will attempt to load traceability information from the earlier build, for which **Model-to-code** must have been selected.

Category: Code Generation > Report

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by “Model-to-code” on page 10-15.

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions”

Eliminated / virtual blocks

Description

Include summary of eliminated and virtual blocks in code generation report.

Category: Code Generation > Report

Settings

Default: On

 On

Includes a summary of eliminated and virtual blocks in the code generation report.

 Off

Does not include a summary of eliminated and virtual blocks.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.

Command-Line Information

Parameter: `GenerateTraceReport`
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

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions”

Traceable Simulink blocks

Description

Include summary of Simulink blocks in code generation report.

Category: Code Generation > Report

Settings

Default: On

 On

Includes a summary of Simulink blocks and the corresponding code location in the code generation report.

 Off

Does not include a summary of Simulink blocks.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.

Command-Line Information

Parameter: `GenerateTraceReportSl`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions”

Traceable Stateflow objects

Description

Include summary of Stateflow objects in code generation report.

Category: Code Generation > Report

Settings

Default: On

 On

Includes a summary of Stateflow objects and the corresponding code location in the code generation report.

 Off

Does not include a summary of Stateflow objects.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.

Command-Line Information

Parameter: `GenerateTraceReportSf`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions”
- “Traceability of Stateflow Objects in Generated Code”

Traceable MATLAB functions

Description

Include summary of MATLAB functions in code generation report.

Category: Code Generation > Report

Settings

Default: On

 On

Includes a summary of MATLAB functions and corresponding code locations in the code generation report.

 Off

Does not include a summary of MATLAB functions.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.

Command-Line Information

Parameter: `GenerateTraceReportEml`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
Traceability	On
Efficiency	No impact
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Code Generation Report” on page 5-2
- “HTML Code Generation Report Extensions”

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**.

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

Related Examples

- "Model Configuration Parameters: Code Generation Report" on page 5-2
- Analyze Code Replacements in the Generated Code
- Trace Code Replacements Generated Using Your Code Replacement Library
- Determine Why Code Replacement Functions Were Not Used

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Target 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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

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`

Specify a maximum word length, in bits, for which the code generation process generates multiword type definitions into the file `multiword_types.h`. All multiword type definitions up to and including this number of bits are generated. If you select 0, multiword type definitions are not generated into the file `multiword_types.h`.

The maximum word length for multiword types only determines the type definitions generated and does not impact the efficiency of the generated code. If the maximum word length for multiword types is set to 0 or too small, an error occurs when the generated code is compiled. This error is caused by the generated code using a type that does not have the required type definition. To resolve the error, increase the maximum word length and regenerate the code. If the maximum word length for multiword types is larger than required, then `multiword_types.h` might contain unused type definitions. Unused type definitions do not consume target resources.

Tips

- Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `multiword_types.h` file during code generation. These updates occur when the new model uses multiword types of length greater than those of the other models. You must then recompile and, depending on your development process, reverify previously generated code. To prevent updates to `multiword_types.h`, determine a maximum word length sufficiently big to cover the needs of all models in the hierarchy. Configure every model in the hierarchy to use that same maximum word length.
- The majority of embedded designs do not need multiword types. By setting maximum word length for multiword types to 0, you can prevent use of multiword variables

on the target. If you use multiword variables with a maximum word length that is 0 or smaller than required, you are alerted with an error when the generated code is compiled.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by selecting the value `User defined` for the parameter **Multiword type definitions**.

Command-Line Information

Parameter: ERTMultiwordLength
Type: integer
Value: valid quantity of bits representing a word size
Default: `256`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

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 target (`grt_main.c` or `grt_main.cpp`) in models created before R2012a.

This option provides a quick way to use code generated in the current release with a GRT-based custom target that has a main program module based on pre-R2012a `grt_main.c` or `grt_main.cpp`.

 Off

Disables the classic call interface.

Tips

The following are unsupported:

- Data type replacement
- Nonvirtual subsystem option **Function with separate data**

Dependencies

- 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**.

- For an ERT target, selecting this option also selects the required option **Support: floating-point numbers**. If you subsequently clear **Support: floating-point numbers**, an error is displayed during code generation.

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Use Discrete and Continuous Time”

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

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No recommendation

See Also

`model_step`

Related Examples

- "Model Configuration Parameters: Advanced Parameters" on page 10-2
- "Entry-Point Functions and Scheduling"
- "Generate Reentrant Code from Top-Level Models"
- "Code Generation of Subsystems"
- "Generate Modular Function Code"

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”.

 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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Configure Code Interface Options”

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. See “Model Blocks and Direct Feed through” for details.

Simulink Coder ignores this parameter for a referenced model if any of the following conditions apply to that model:

- Is multi-rate
- Has a continuous sample time
- Is logging states (using the **States** or **Final states** parameters in the **Configuration Parameters** > **Data Import/Export** pane

Dependencies

- 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
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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “rt_OneStep and Scheduling Considerations”

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

Application	Setting
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	No recommendation

See Also

`model_terminate`

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

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

Application	Setting
Traceability	No impact
Efficiency	On
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Global Block I/O Structure”
- “Storage Classes for Block States”

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Configure Code Interface Options”

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Configure Code Interface Options”

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Configure Code Interface Options”

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")
- `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" 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**.
- For the GRT target, selecting this option also selects the required option **Support non-finite numbers**. If you subsequently clear **Support non-finite numbers**, an error is displayed during code generation.
- For ERT-based targets, selecting this option also selects the required options **Support: floating-point numbers**, **Support: non-finite numbers**, and **Terminate function required**. If you subsequently clear **Support: floating-point numbers**, **Support: non-finite numbers**, or **Terminate function required**, an error is displayed during code generation.
- 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

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 storage classes and custom storage classes, for example, by using the Model Data Editor. 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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Log Program Execution Results”
- “Log Data for Analysis”
- “Virtualized Output Ports Optimization”
- “Virtualized Output Ports Optimization”

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Log Program Execution Results”
- “Log Data for Analysis”

Verbose build

Description

Display code generation progress.

Category: Code Generation > Debug

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

- “Debug”

Retain .rtw file

Description

Specify *model*`.rtw` file retention.

Category: Code Generation > Debug

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

- "Debug"

Profile TLC

Description

Profile the execution time of TLC files.

Category: Code Generation > Debug

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

- "Debug"

Start TLC debugger when generating code

Description

Specify use of the TLC debugger

Category: Code Generation > Debug

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
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Debug”

Start TLC coverage when generating code

Description

Generate the TLC execution report.

Category: Code Generation > Debug

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
Efficiency	No impact

Application	Setting
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Debug”

Enable TLC assertion

Description

Produce the TLC stack trace

Category: Code Generation > Debug

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

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

- “Debug”

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

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2
- “Speed Up Linear Algebra in Code Generated from a MATLAB Function Block”

Use Simulink Coder Features

Description

Enable “Simulink Coder” features for models deployed to “Simulink Supported Hardware”.

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 “Model Configuration Parameters: Advanced Parameters” on page 10-2.

 Off

Disable the “Model Configuration Parameters: Advanced Parameters” on page 10-2.

Indicates that this parameter is enabled. To disable it, first disable the “Use Embedded Coder Features” parameter.

Dependencies

This parameter requires a Simulink Coder or Embedded Coder license.

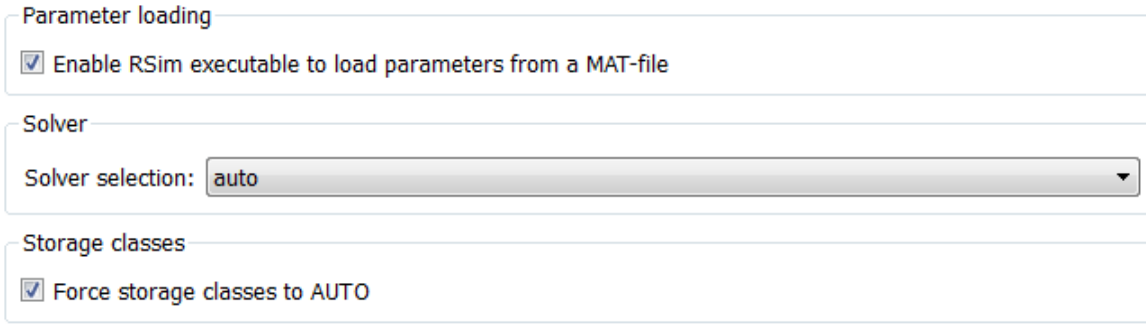
Related Examples

- “Model Configuration Parameters: Advanced Parameters” on page 10-2

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 4-5.

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

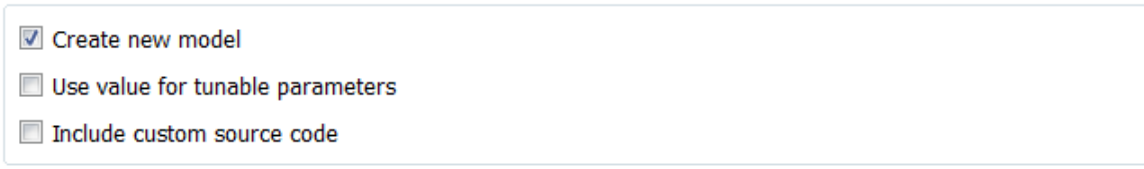
Parameter: `RSIM_STORAGE_CLASS_AUTO`
Type: character vector
Value: `'on' | 'off'`
Default: `'on'`

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 4-5.

See Also

- "Accelerate Simulation, Reuse Code, or Protect Intellectual Property by Using S-Function Target"
- "Code Generation Pane: S-Function Target" on page 11-8

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

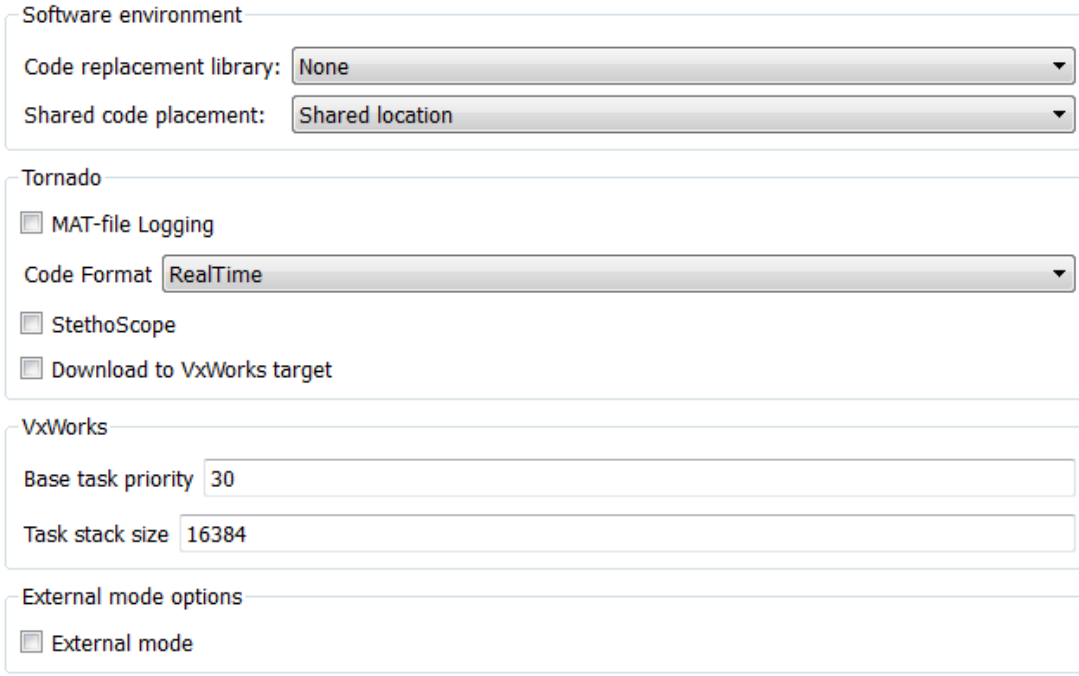
Parameter: `AlwaysIncludeCustomSrc`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'`

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 4-5.

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 11-14

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

Application	Setting
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Target 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.

For more information about selections for this parameter, see "Code replacement library" on page 9-6.

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

"Target 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

- "Target 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")
- `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" 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"

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

"System Target Files and Code Generation Features"

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
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

"Set Up and Use Host/Target Communication Channel"

Transport layer

Specify the transport protocol for External mode communications.

Settings

Default: `tcpip`

`tcpip`

Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. For targets provided by MathWorks, the value is specified in *`matlabroot`*`/toolbox/simulink/simulink/extmode_transports.m`.

Dependency

This parameter is enabled by the **External mode** check box.

Command-Line Information

Parameter: `ExtModeTransport`
Type: integer
Value: `0`
Default: `0`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Target Interfacing"

MEX-file arguments

Specify arguments to pass to an External mode interface MEX-file for communicating with executing targets.

Settings

Default: `''`

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target (for example, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

Dependency

This parameter is enabled by the **External mode** check box.

Command-Line Information

Parameter: `ExtModeMexArgs`
Type: 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

- "Target Interfacing"
- "Choose Communication Protocol for Client and Server"

Static memory allocation

Control the memory buffer for External mode communication.

Settings

Default: off

 On

Enables the **Static memory buffer size** parameter for allocating allocate dynamic memory.

 Off

Uses a static memory buffer for External mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- This parameter is enabled by the **External mode** check box.
- This parameter enables **Static memory buffer size**.

Command-Line Information

Parameter: `ExtModeStaticAlloc`
Type: 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

“Configure External Mode Options for Code Generation”

Static memory buffer size

Specify the memory buffer size for External mode communication.

Settings

Default: 1000000

Enter the number of bytes to preallocate for External mode communications buffers in the target.

Tips

- If you enter too small a value for your application, External mode issues an out-of-memory error.
- To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information

Parameter: `ExtModeStaticAllocSize`
Type: integer
Value: valid value
Default: 1000000

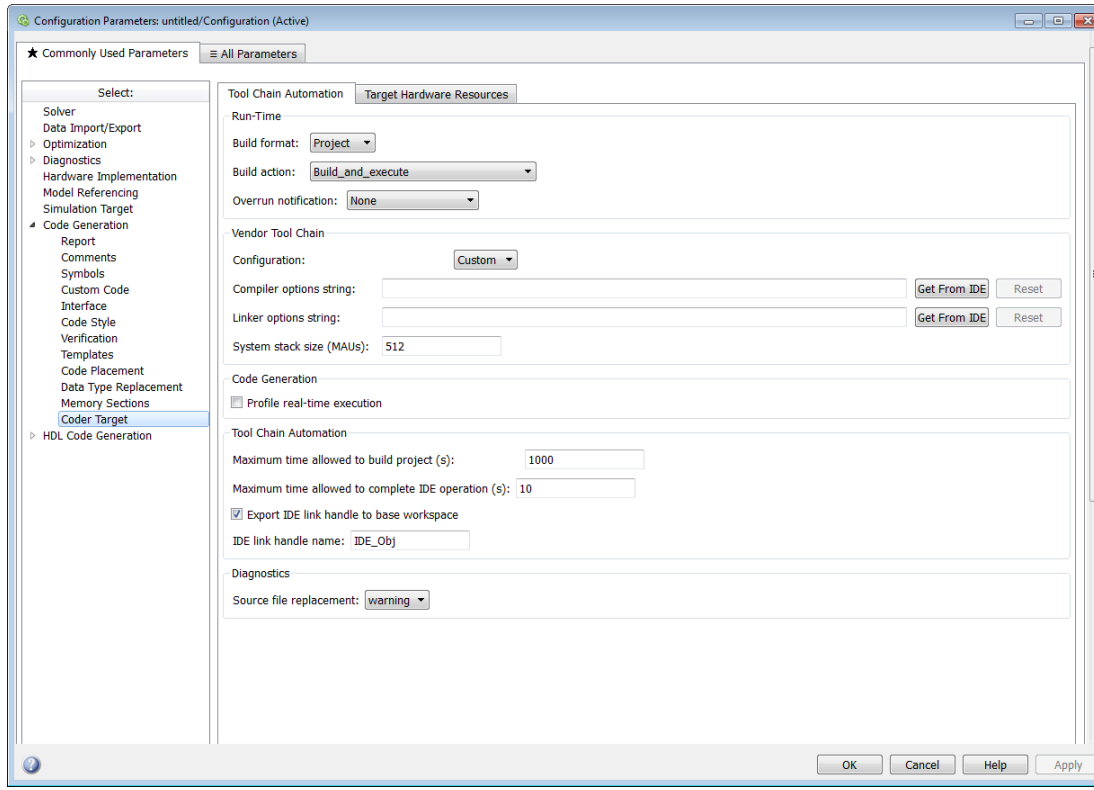
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure External Mode Options for Code Generation"

Code Generation: Coder Target Pane



In this section...

In this section...

Code Generation: Coder Target Pane Overview (previously "IDE Link Tab Overview")

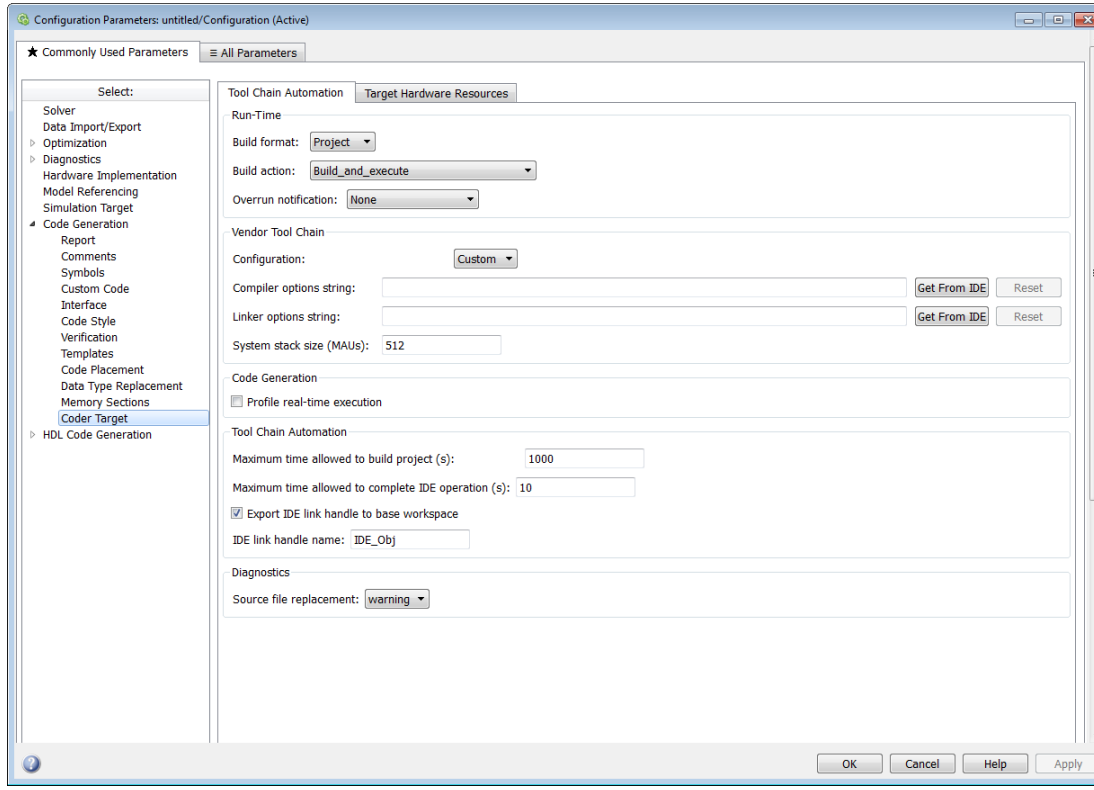
Configure the parameters for:

- Tool Chain Automation — How the code generator interacts with third-party software build toolchains.
- Target Hardware Resources — The IDE toolchain and properties of the physical hardware, such as board, operating system, memory, and peripherals.

See Also

- "Coder Target: Tool Chain Automation Tab Overview"
- "Coder Target: Target Hardware Resources Tab Overview"

Coder Target: Tool Chain Automation Tab Overview



The Tool Chain Automation Tab is only visible under the Coder Target pane.

The following table lists the parameters on the Tool Chain Automation Tab.

- "Build format" on page 11-45
- "Build action" on page 11-47
- "Overrun notification" on page 11-50
- "Function name" on page 11-52
- "Configuration" on page 11-53
- "Compiler options string" on page 11-55
- "Linker options string" on page 11-56

Build format

Defines how the code generator responds when you press Ctrl+B to build your model.

Settings

Default: `Project`

`Project`
: Builds your model as an IDE project.

`Makefile`
: Creates a makefile and uses it to build your model.

Dependencies

Selecting `Makefile` removes the following parameters:

- **Code Generation**
  - **Profile real-time execution**
  - **Profile by**
  - **Number of profiling samples to collect**
- **Link Automation**
  - **Maximum time allowed to build project (s)**
  - **Maximum time allowed to complete IDE operation (s)**
  - **Export IDE link handle to base workspace**
  - **IDE link handle name**

Command-Line Information

Parameter: `buildFormat`
Type: character vector
Value: `'Project' | 'Makefile'`
Default: `'Build_and_execute'`

Recommended Settings

Application	Setting
Debugging	`Project`

Application	Setting
Traceability	`Project`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Build action

Defines how the code generator responds when you press Ctrl+B to build your model.

Settings

Default: `Build_and_execute`

If you set **Build format** to `Project`, select one of the following options:

`Build_and_execute`

Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

`Create_project`

Directs the code generator to create a new project in the IDE. The command line equivalent for this setting is `Create`.

`Archive_library`

Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

`Build`

Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

`Create_processor_in_the_loop_project`

Directs the code generator to create PIL algorithm object code as part of the project build.

If you set **Build format** to `Makefile`, select one of the following options:

`Create_makefile`

Creates a makefile. For example, ".mk". The command line equivalent for this setting is `Create`.

`Archive_library`

Creates a makefile and an archive library. For example, ".a" or ".lib".

`Build`

Creates a makefile and an executable. For example, ".exe".

`Build_and_execute`

Creates a makefile and an executable. Then it evaluates the execute instruction under the **Execute** tab in the current XMakefile configuration.

Dependencies

Selecting `Archive_library` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace**

Selecting `Create_processor_in_the_loop_project` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

Command-Line Information

Parameter: `buildAction`
Type: character vector
Value: `'Build' | 'Build_and_execute' | 'Create' | 'Archive_library' | 'Create_processor_in_the_loop_project'`
Default: `'Build_and_execute'`

Recommended Settings

Application	Setting
Debugging	`Build_and_execute`
Traceability	`Archive_library`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

Overrun notification

Specifies how your program responds to overrun conditions during execution.

Settings

Default: `None`

`None`

Your program does not notify you when it encounters an overrun condition.

`Print_message`

Your program prints a message to standard output when it encounters an overrun condition.

`Call_custom_function`

When your program encounters an overrun condition, it executes a function that you specify in **Function name**.

Tips

- The definition of the standard output depends on your configuration.

Dependencies

Selecting `Call_custom_function` enables the **Function name** parameter.

Setting this parameter to `Call_custom_function` enables the **Function name** parameter.

Command-Line Information

Parameter: `overrunNotificationMethod`
Type: character vector
Value: `'None' | 'Print_message' | 'Call_custom_function'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Print_message` or `Call_custom_function`
Traceability	`Print_message`

Application	Setting
Efficiency	None
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Function name

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

Settings

No Default

Dependencies

This parameter is enabled by setting **Overrun notification** to `Call_custom_function`.

Command-Line Information

Parameter: `overrunNotificationFcn`
Type: character vector
Value: no default
Default: no default

Recommended Settings

Application	Setting
Debugging	String
Traceability	String
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Configuration

Sets the Configuration for building your project from the model.

Settings

Default: `Custom`

`Custom`

Lets the user apply a specialized combination of build and optimization settings.

`Custom` applies the same settings as the Release project configuration in IDE, except:

- The compiler options do not use optimizations.
- The memory configuration specifies a memory model that uses `Far Aggregate` for data and `Far` for functions.

`Debug`

Applies the `Debug` Configuration defined by the IDE to the generated project and code.

`Release`

Applies the `Release` project configuration defined by the IDE to the generated project and code.

Dependencies

- Selecting `Custom` disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting `Release` sets the **Compiler options string** to the settings defined by the IDE.
- Selecting `Debug` sets the **Compiler options string** to the settings defined by the IDE.

.

Command-Line Information

Parameter: `projectOptions`
Type: character vector
Value: `'Custom' | 'Debug' | 'Release'`
Default: `'Custom'`

Recommended Settings

Application	Setting
Debugging	`Custom` or `Debug`
Traceability	`Custom`, `Debug`, `Release`
Efficiency	`Release`
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the code generator does not set optimization flags.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Configuration** to `Custom` applies the `Custom` compiler options defined by the code generator. `Custom` does not use optimizations.
- Setting **Configuration** to `Debug` applies the debug settings defined by the IDE.
- Setting **Configuration** to `Release` applies the release settings defined by the IDE.

Command-Line Information

Parameter: `compilerOptionsStr`
Type: character vector
Value: `'Custom' | 'Debug' | 'Release'`
Default: `'Custom'`

Recommended Settings

Application	Setting
Debugging	`Custom`
Traceability	`Custom`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Linker options string

To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, the code generator does not set linker options.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information

Parameter: `linkerOptionsStr`
Type: character vector
Value: valid linker option
Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory.

This parameter is used in targets to allocate the stack size for the generated application. For example, with embedded processors that are not running an operating system, this parameter determines the total stack space that can be used for the application. For operating systems such as Linux or Windows , this value specifies the stack space allocated per thread.

This parameter also affects the "Maximum stack size (bytes)" parameter, located in the Optimization > Signals and Parameters pane.

Settings

Default: 8192

Minimum: 0

Maximum: Available memory

- Enter the stack size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.
- The software does not verify the value you entered is valid.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

When you set the **System target file** parameter on the **Code Generation** pane to `idelink_ert.tlc` or `idelink_grt.tlc`, the software sets the **Maximum stack size** parameter on the **Optimization** > **Signals and Parameters** pane to `Inherit from target` and makes it non-editable. In that case, the **Maximum stack size** parameter compares the value of (**System stack size**/2) with 200,000 bytes and uses the smaller of the two values.

Command-Line Information

Parameter: `systemStackSize`
Type: int
Default: `8192`

Recommended Settings

Application	Setting
Debugging	int
Traceability	int
Efficiency	int
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Profile real-time execution

Enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

Settings

Default: Off

 On

Adds instrumentation to the generated code to support execution profiling and generate the profiling report.

 Off

Does not instrument the generated code to produce the profile report.

Dependencies

This parameter adds **Number of profiling samples to collect** and **Profile by**.

Selecting this parameter enables **Export IDE link handle to base workspace** and makes it non-editable, since the code generator must create a handle.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop project` removes this parameter.

Command-Line Information

Parameter: `ProfileGenCode`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics..

Profile by

Defines which execution profiling technique to use.

Settings

Default: `Task`

`Task`

Profiles model execution by the tasks in the model.

`Atomic subsystem`

Profiles model execution by the atomic subsystems in the model.

Dependencies

Selecting **Real-time execution profiling** enables this parameter.

Command-Line Information

Parameter: `profileBy`
Type: character vector
Value: `Task | Atomic subsystem`
Default: `Task`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics.

Number of profiling samples to collect

Specify the size of the buffer that holds the profiling samples. Enter a value that is 2 times the number of profiling samples.

Each task or subsystem execution instance represents one profiling sample. Each sample requires two memory locations, one for the start time and one for the end time. Consequently, the size of the buffer is twice the number of samples.

Sample collection begins with the start of code execution and ends when the buffer is full.

The profiling data is held in a statically sited buffer on the target processor.

Settings

Default: 100

Minimum: 2

Maximum: Buffer capacity

Tips

- Data collection stops when the buffer is full, but the application and processor continue running.
- Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

Dependencies

This parameter is enabled by **Profile real-time execution**.

Command-Line Information

Parameter:`ProfileNumSamples`
Type: int
Value: Positive integer
Default: `100`

Recommended Settings

Application	Setting
Debugging	100

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

Settings

Default: 1000

Minimum: 1

Maximum: No limit

Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to complete IDE operation** timeout value.

Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

Command-Line Information

Parameter:`ideObjBuildTimeout`
Type: int
Value: Integer greater than 0
Default: `100`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Maximum time allowed to complete IDE operation (s)

specifies how long, in seconds, the software waits for IDE functions, such as `read` or `write`, to return completion messages.

Settings

Default: 10

Minimum: 1

Maximum: No limit

Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to build project (s)** timeout value

Command-Line Information

Parameter: `'ideObjTimeout'`
Type: int
Value:
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Export IDE link handle to base workspace

Directs the software to export the `IDE_Obj` object to your MATLAB workspace.

Settings

Default: On

 On

Directs the build process to export the `IDE_Obj` object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.

 Off

prevents the build process from exporting the `IDE_Obj` object to your MATLAB software workspace.

Dependency

Selecting **Profile real-time execution** enables **Export IDE link handle to base workspace** and makes it non-editable, since the code generator must create a handle.

Selecting **Export IDE link handle to base workspace** enables **IDE link handle name**.

Command-Line Information

Parameter: `exportIDEObj`
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

For more information, refer to the “Code Generation Pane: Coder Target” topic.

IDE link handle name

specifies the name of the `IDE_Obj` object that the build process creates.

Settings

Default: `IDE_Obj`

- Enter a valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the `IDE_Obj` object.
- The handle name is case sensitive.

Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

Command-Line Information

Parameter: `ideObjName`
Type: character vector
Value:
Default: `IDE_Obj`

Recommended Settings

Application	Setting
Debugging	Enter a valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Source file replacement

Selects the diagnostic action to take if the code generator detects conflicts that you are replacing source code with custom code.

Settings

Default: `warn`

`none`

Does not generate warnings or errors when it finds conflicts.

`warning`

Displays a warning.

`error`

Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

Tips

- The build operation continues if you select `warning` and the software detects custom code replacement. You see warning messages as the build progresses.
- Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select `none` when you do not want to see multiple messages during your build.
- The messages apply to code generator **Custom Code** replacement options as well.

Command-Line Information

Parameter: `DiagnosticActions`
Type: character vector
Value: `none | warning | error`
Default: `warning`

Recommended Settings

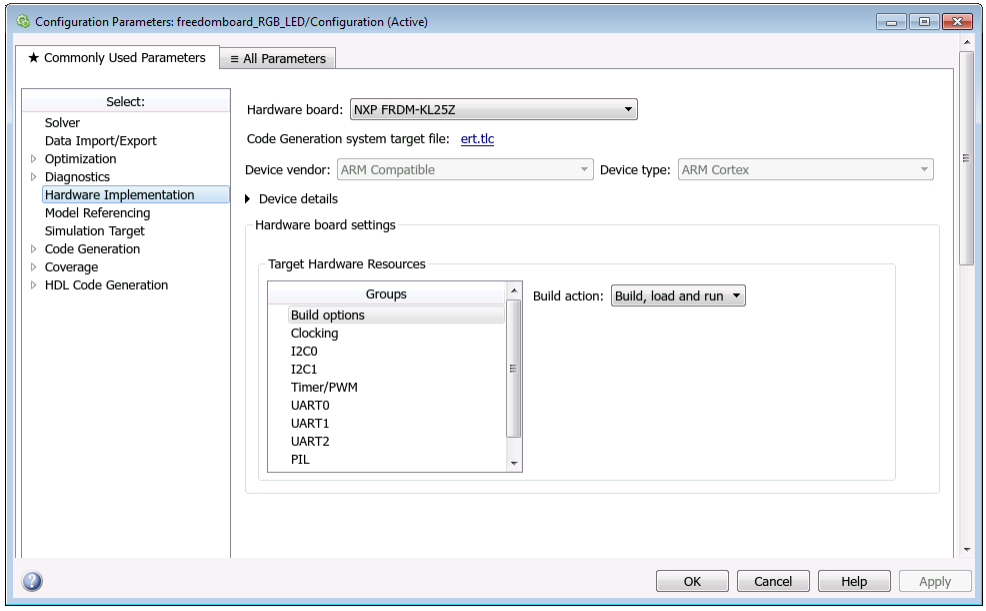
Application	Setting
Debugging	`error`
Traceability	`error`

Application	Setting
Efficiency	`warning`
Safety precaution	`error`

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Hardware Implementation Pane



In this section...

Code Generation Pane

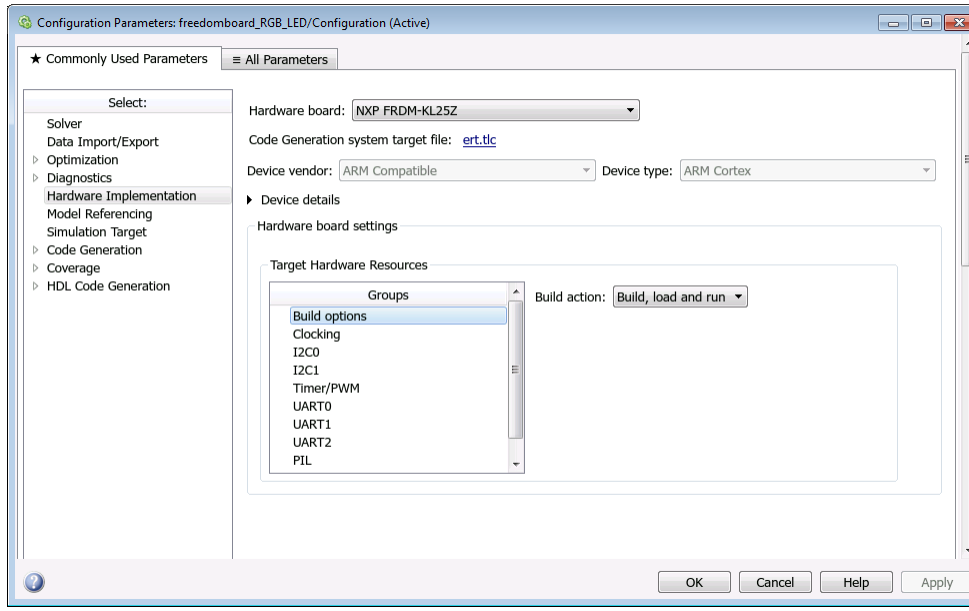
Select the system target file and the toolchain for your hardware.

Scheduler options

Scheduler interrupt source

Select the source of the scheduler interrupt.

Build Options



To specify how the build process takes place during code generation, select build options.

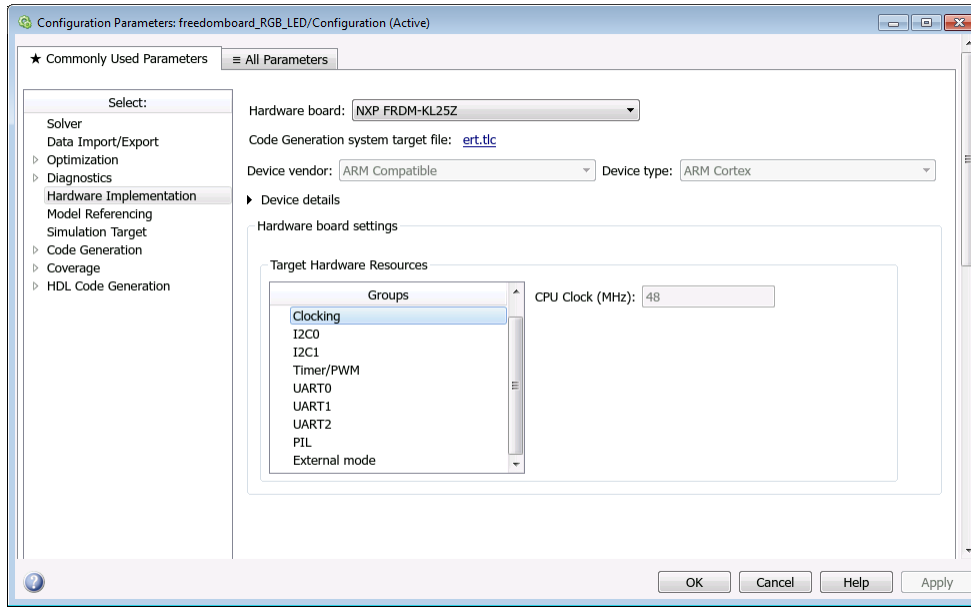
Build action

Specify whether you want only a build action or build, load, and run actions during code generation.

Default: `Build, load and run`

- `Build` — Select this option if you want to build the code during the build process.
- `Build, load and run` — Select this option to build, load, and run the generated code during the build process.

Clocking

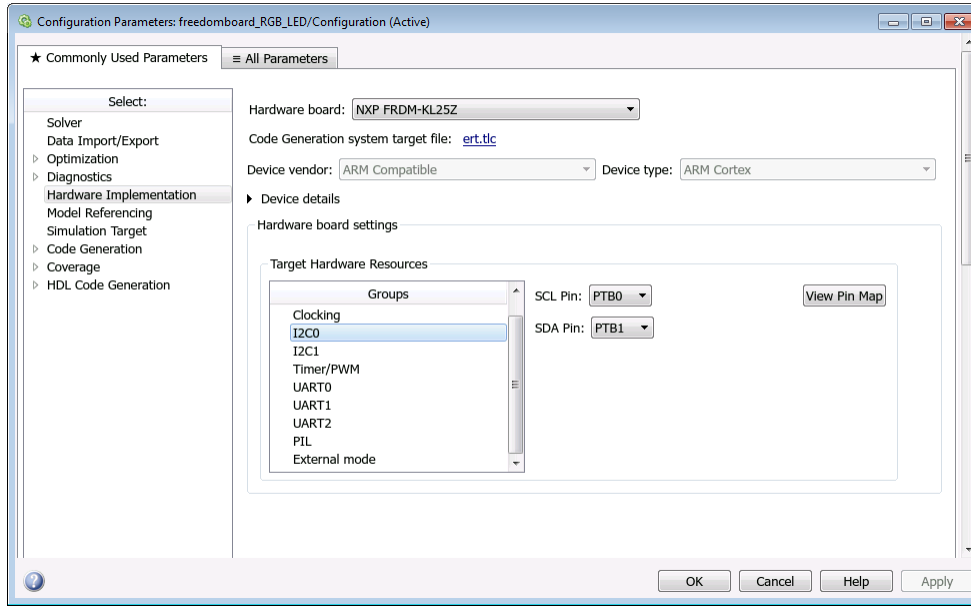


CPU Clock (MHz)

This option is for the CPU clock frequency of the FRDM-KL25Z processor on the target hardware.

Note: This parameter appears dimmed. The value of this parameter is set to 48 MHz.

I2C0 and I2C1



SCL Pin

Select an SCL pin for I2C communication.

I2C0

Default: `PTB0`

`PTB0, PTB2, PTC8, PTE24`

I2C1

Default: `PTE1`

`PTE1, PTC1, PTC10`

SDA Pin

Select an SDA pin for I2C communication.

I2C0

Default: PTB1

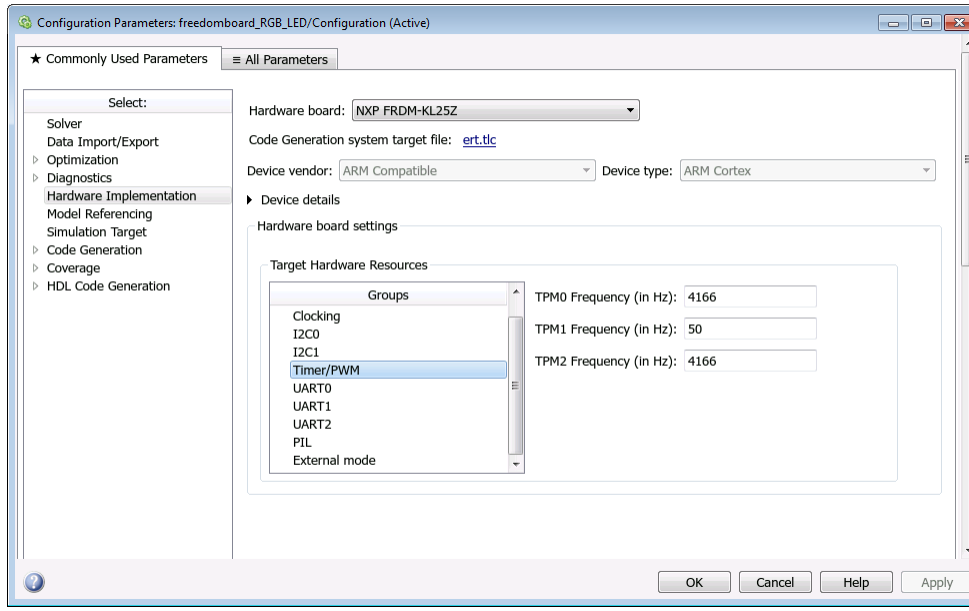
PTB1, PTB3, PTC9, PTE25

I2C1

Default: PTE0

PTE0, PTC2, PTC11, PTA4

Timer/PWM



TPM0 Frequency (in Hz)

Specify the frequency for the TPM0 timer.

Default: 4166 Hz

TPM1 Frequency (in Hz)

Specify the frequency for the TPM1 timer.

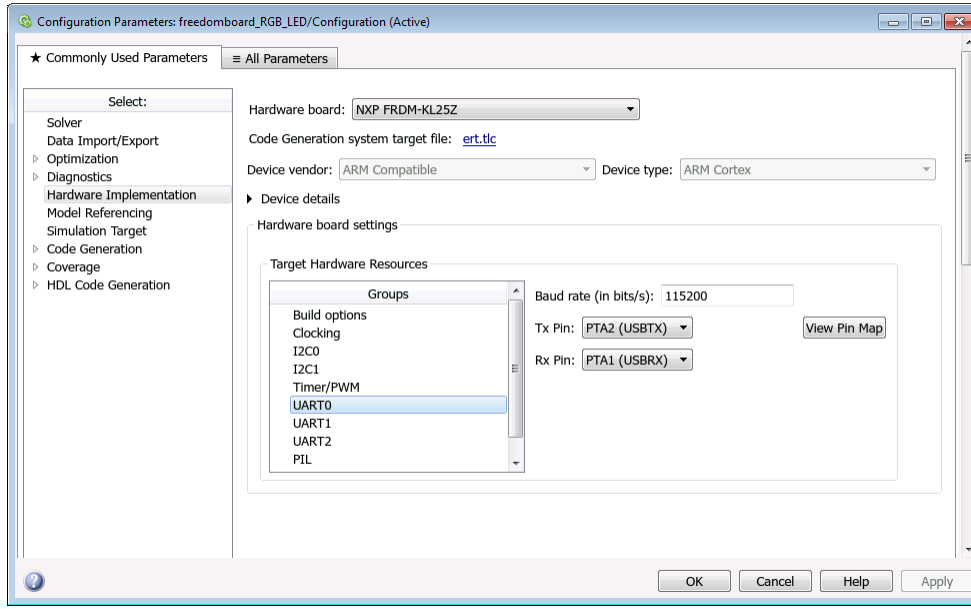
Default: 50 Hz

TPM2 Frequency (in Hz)

Specify the frequency for the TPM2 timer.

Default: 4166 Hz

UART0, UART1, and UART2



Baud rate (in bits/s)

Specify the baud for UARTx serial interfaces.

Default: `115200`

Tx Pin

Select a Tx pin for serial communication.

UART0

Default: `PTA2 (USBTX)`

`PTA2 (USBTX), PTE20, PTD7, No connection`

UART1

Default: `PTC4`

`PTC4, PTE0, No connection`

UART2

Default: `PTD5`

`PTD5, PTE22, PTD3, No connection`

Rx Pin

Select an Rx pin for serial communication.

UART0

Default: `PTA1 (USBRX)`

`PTA1 (USBRX), PTE21, PTD6, No connection`

UART1

Default: `PTC3`

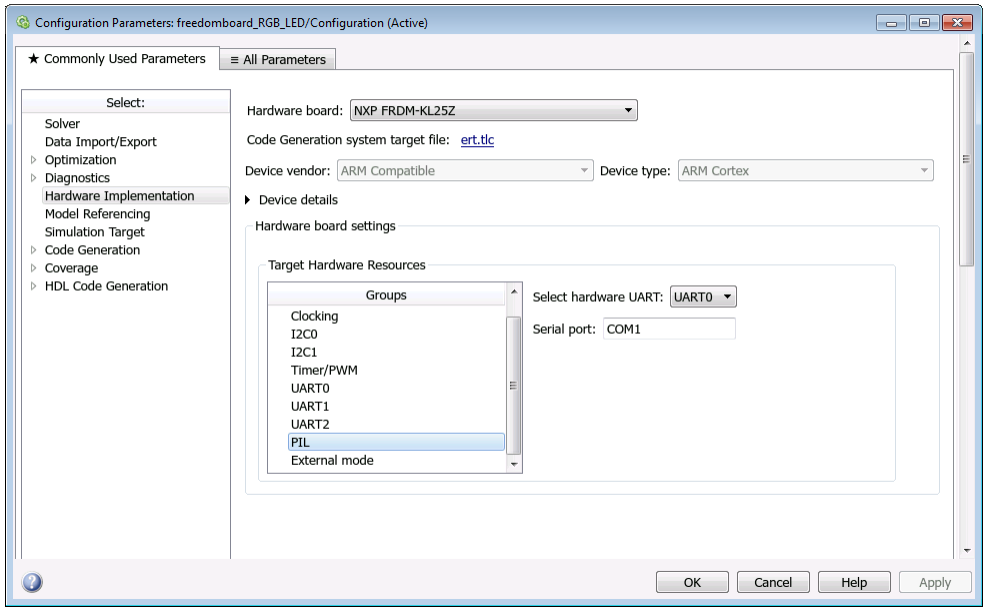
`PTC3, PTE1, No connection`

UART2

Default: `PTD4`

`PTD4, PTE23, PTD2, No connection`

PIL



Select hardware UART

Select the target UART port for PIL communication. After selecting an UART port, go to the selected UART in **Configuration Parameters > Hardware Implementation pane > UARTx** and select the Tx and the Rx pins.

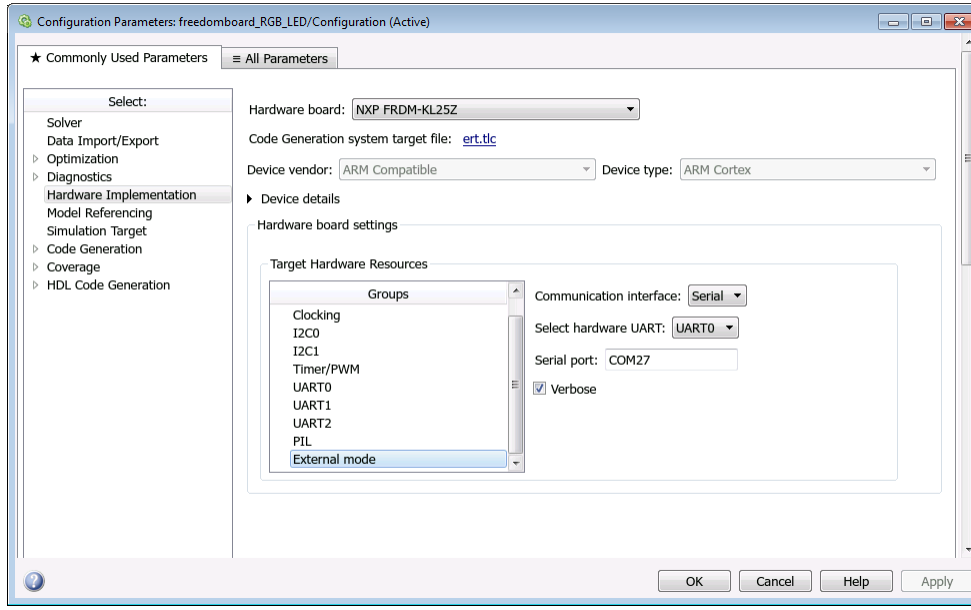
Default: `UART0`

Serial port

Enter the serial port for PIL communication.

Default: `COM1`

External mode



Communication interface

Use the serial option to run your model in the external mode with serial communication.

Default: `serial`

Select hardware UART

Select the target UART port for external mode communication. After selecting an UART port, go to the selected UART in **Configuration Parameters > Hardware Implementation pane > UARTx** and select the Tx and the Rx pins.

Default: `UART0`

Note: The target UART0 port for external mode communication works only in Windows and Mac OS X El Capitan operating systems.

Serial port

Enter the serial port for external mode communication.

Default: `COM27`

Verbose

Select this check box to view the external mode execution progress and updates in the Diagnostic Viewer or in the MATLAB Command Window.

Hardware Implementation Pane

In this section...

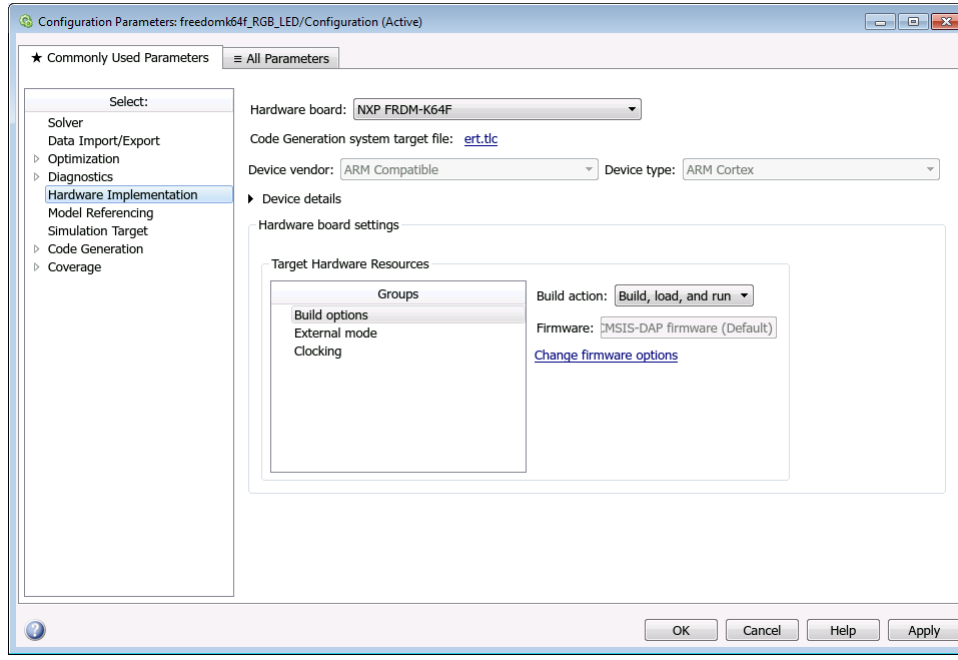
Hardware Implementation Pane Overview

Specify the options for creating and running applications on target hardware.

Configuration

Configure hardware board to run Simulink model.

1. In the Simulink Editor, select **Simulation** > **Model Configuration Parameters**.
2. In the Configuration Parameter dialog box, click **Hardware Implementation**.

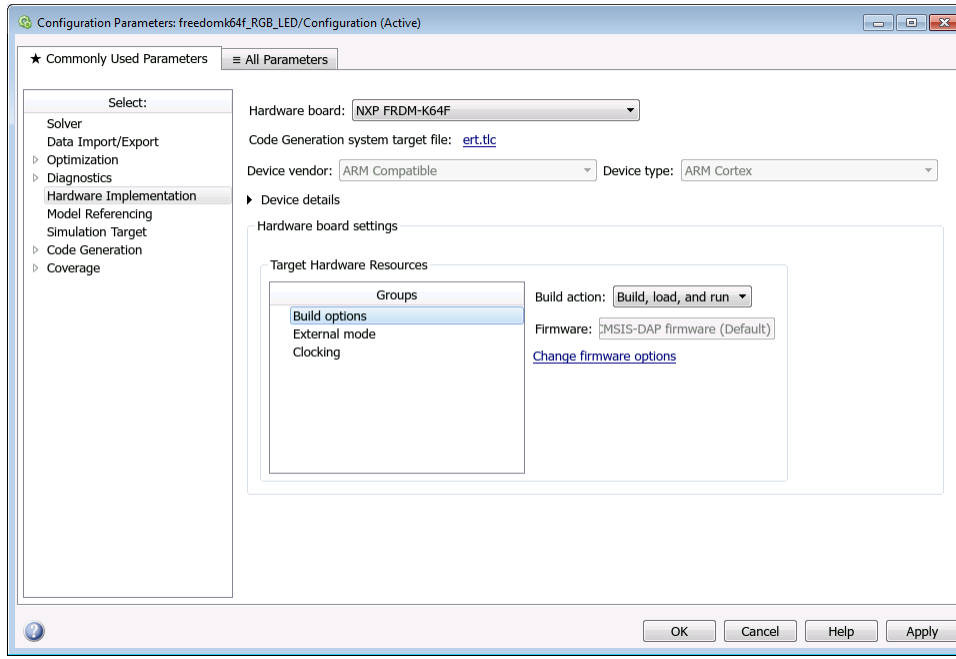


3. Set the **Hardware board** parameter to match your target hardware board.
4. Apply the changes.

Base rate task priority

The value in this parameter defines the priority of the base rate task.

Build options



To specify how the build process takes place during code generation, select build options.

Build action

Specify whether you want only a build action or build, load, and run actions during code generation.

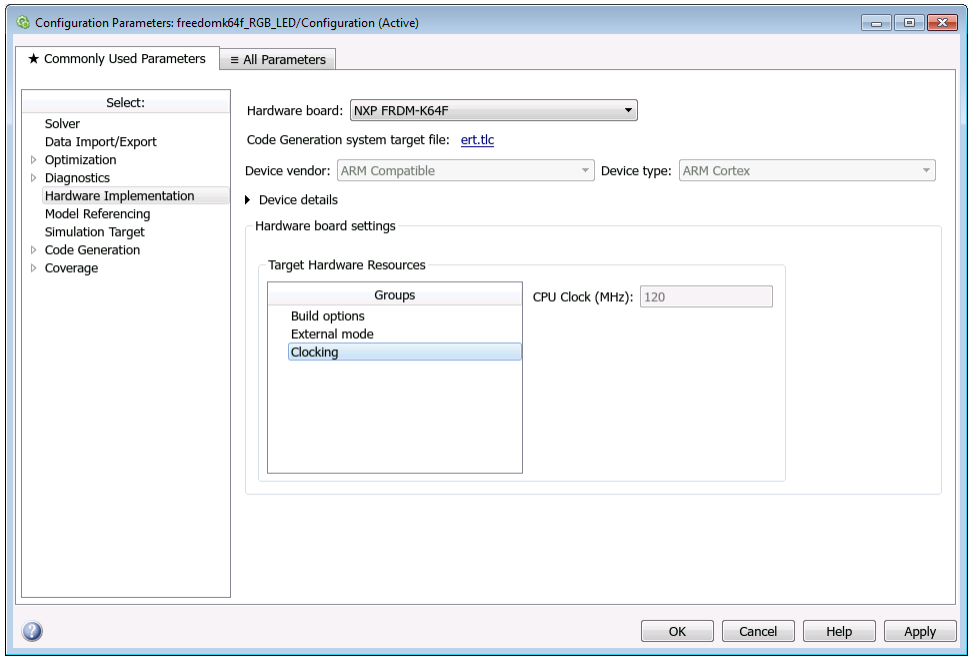
Default: `Build, load and run`

- `Build` — Select this option if you want to build the code during the build process.
- `Build, load and run` — Select this option to build, load, and run the generated code during the build process.

Firmware

This is the firmware chosen during the setup to run your model on the FRDM-K64F board.

Clocking

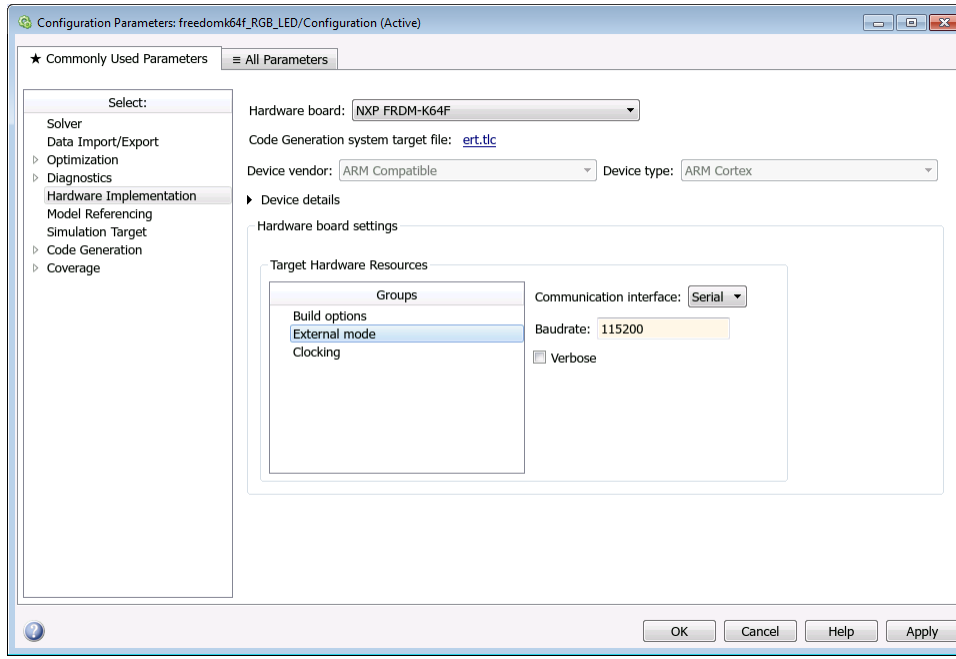


CPU Clock (MHz)

This option is for the CPU clock frequency of the FRDM-K64F processor on the target hardware.

Note: This parameter appears dimmed. The value of this parameter is set to 120 MHz.

External mode



Communication interface

Use the serial option to run your model in the external mode with serial communication.

Default: `Serial`

Baudrate

Specify the baud for UARTx serial interfaces.

Default: `115200`

Verbose

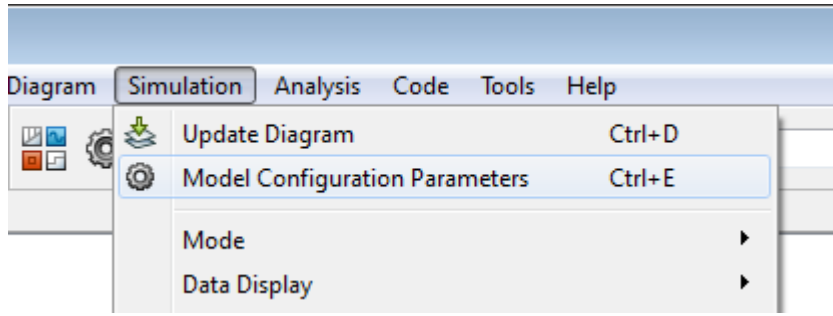
Select this check box to view the external mode execution progress and updates in the Diagnostic Viewer or in the MATLAB Command Window.

Hardware Implementation Pane

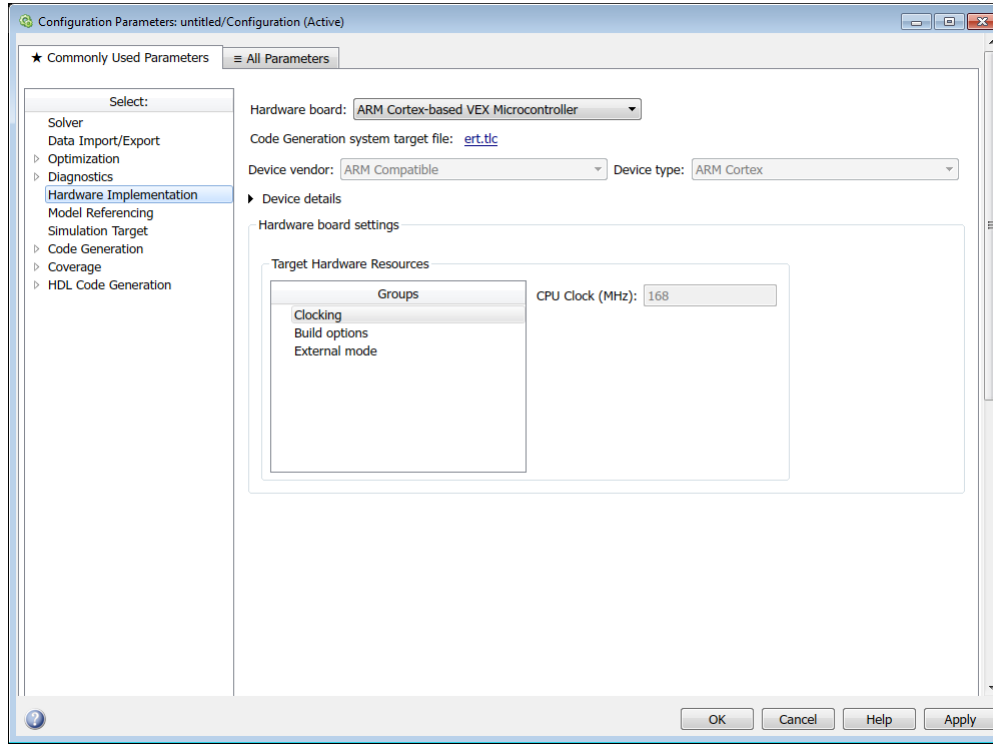
Hardware Implementation Pane Overview

Specify the options for creating and running applications on target hardware.

1 In the Simulation Editor, select **Simulation** > **Model Configuration Parameters**.



2 In the Configuration Parameters dialog box, click **Hardware Implementation**.



3 Select the **Hardware board** to match your VEX® board.

4 Set the target hardware resource parameters.

5 Click **Apply**.

Hardware board

Select the type of hardware upon which to run your model.

Changing this parameter updates the Configuration Parameters dialog, so it only displays parameters that are relevant to your target hardware.

After installing support for your target hardware, reopen the Configuration Parameters dialog and select your target hardware.

To run the model on your VEX device, select `ARM Cortex-based VEX Microcontroller`.

Settings

Default: `None`

`None`

This setting means your model has not been configured to run on target hardware. Choose your target hardware from the list of options.

`Get more...`

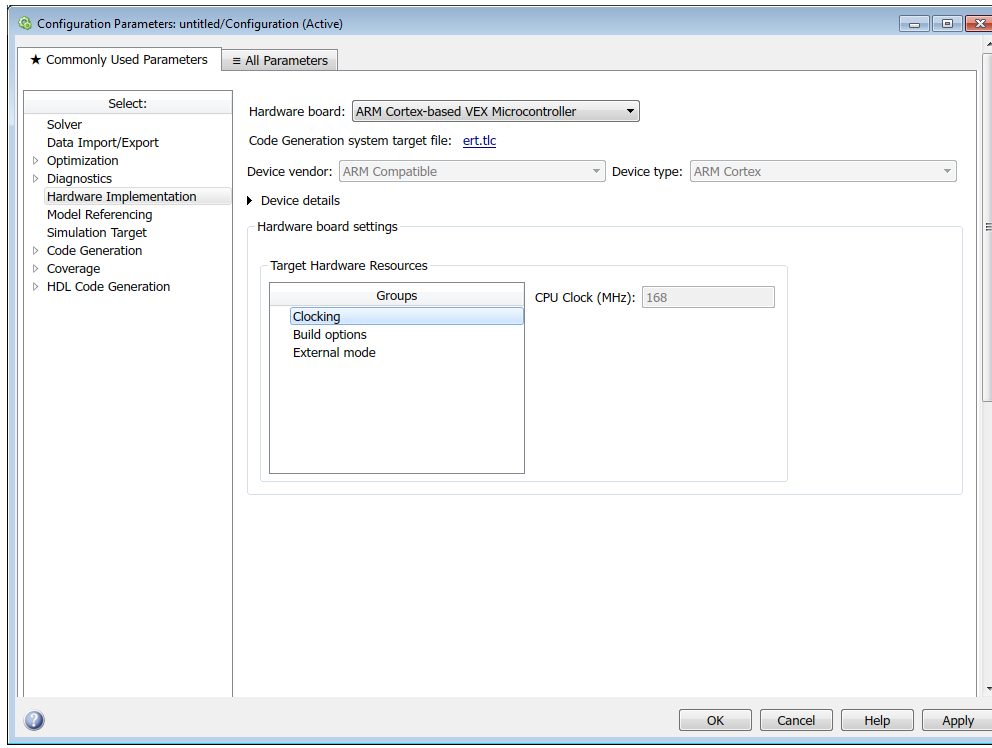
Select this option to start Support Package Installer and install support for additional hardware.

Base rate task priority

Base rate task priority

This parameter sets the static priority of the base rate task. By default, the priority is 40.

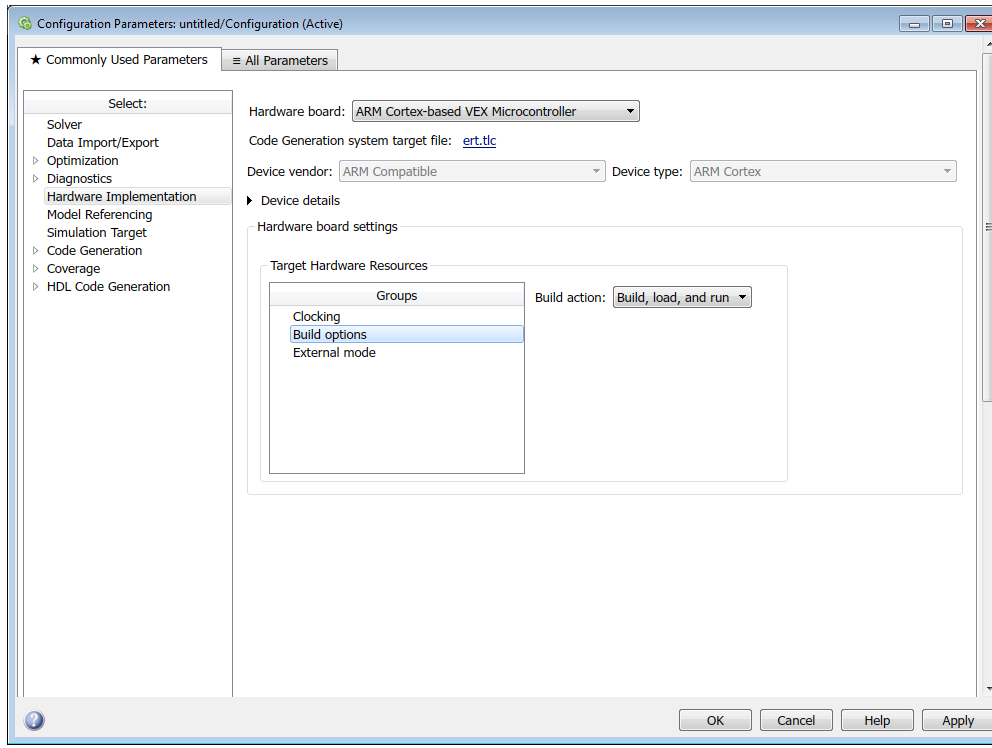
Clocking



CPU Clock (MHz)

This is the CPU clock rate.

Build options

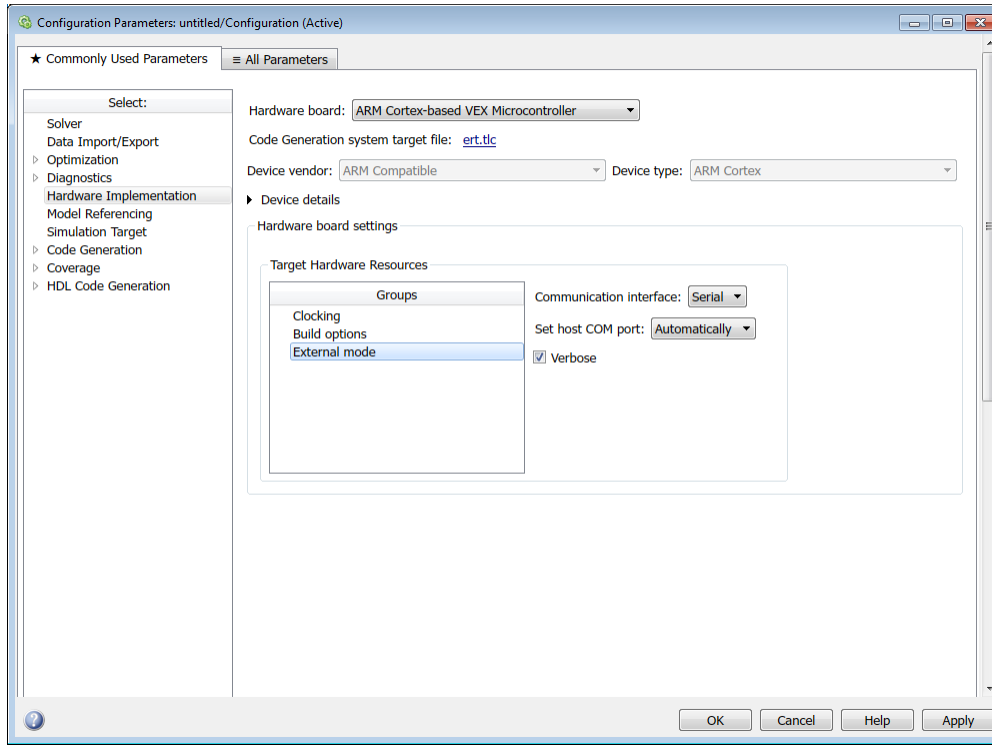


Build action

This is the option to specify, if you want only build action, or build, load and run actions during code generation.

- `Build, load and run` — Select this option to build, load, and to run the generated code.
- `Build` — Select this option to only build the code.

External mode



Communication interface

The 'serial' option uses serial communication for external mode.

Set host COM port

- `Automatically` — Select this option to set the host COM port automatically. This is the default option.
- `Manually` — Select this option to manually set the host COM port.

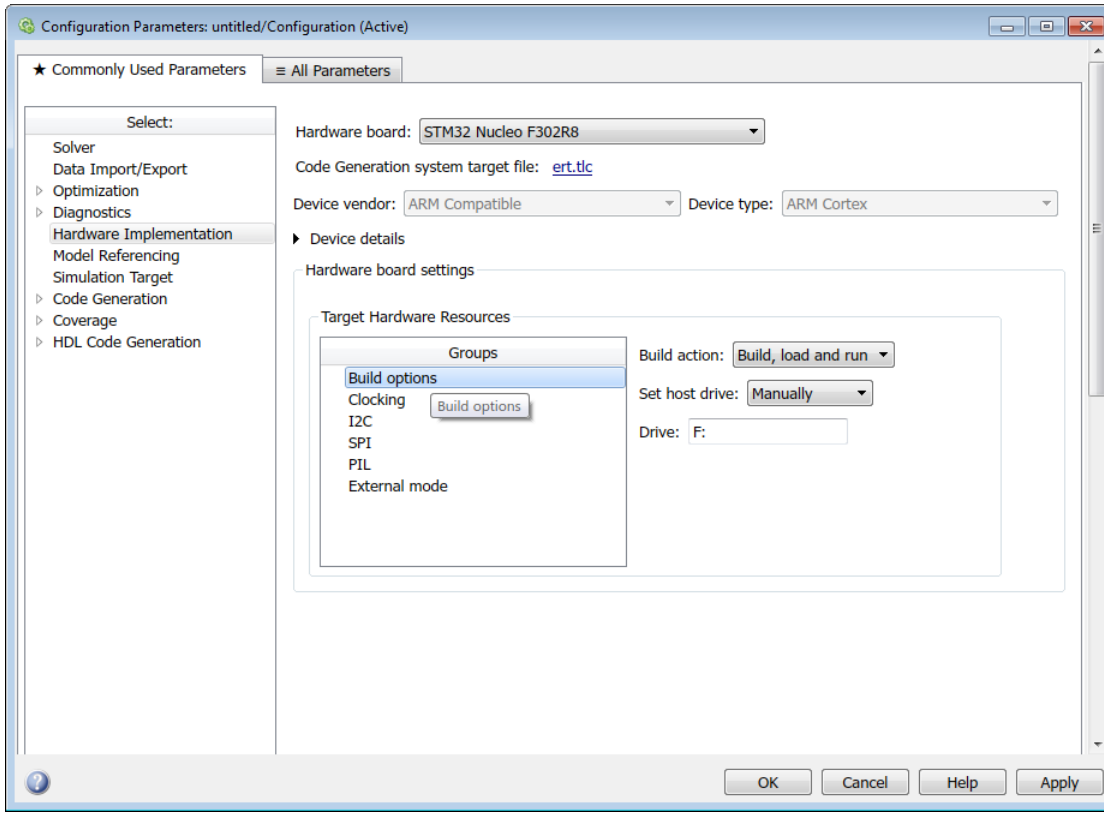
COMPort

Enter the COMPort used by the target hardware. This parameter appears, only if you have selected `Manually` option in the **Set host COM port** parameter.

Verbose

Select this check box to view the External Mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

Hardware Implementation Pane

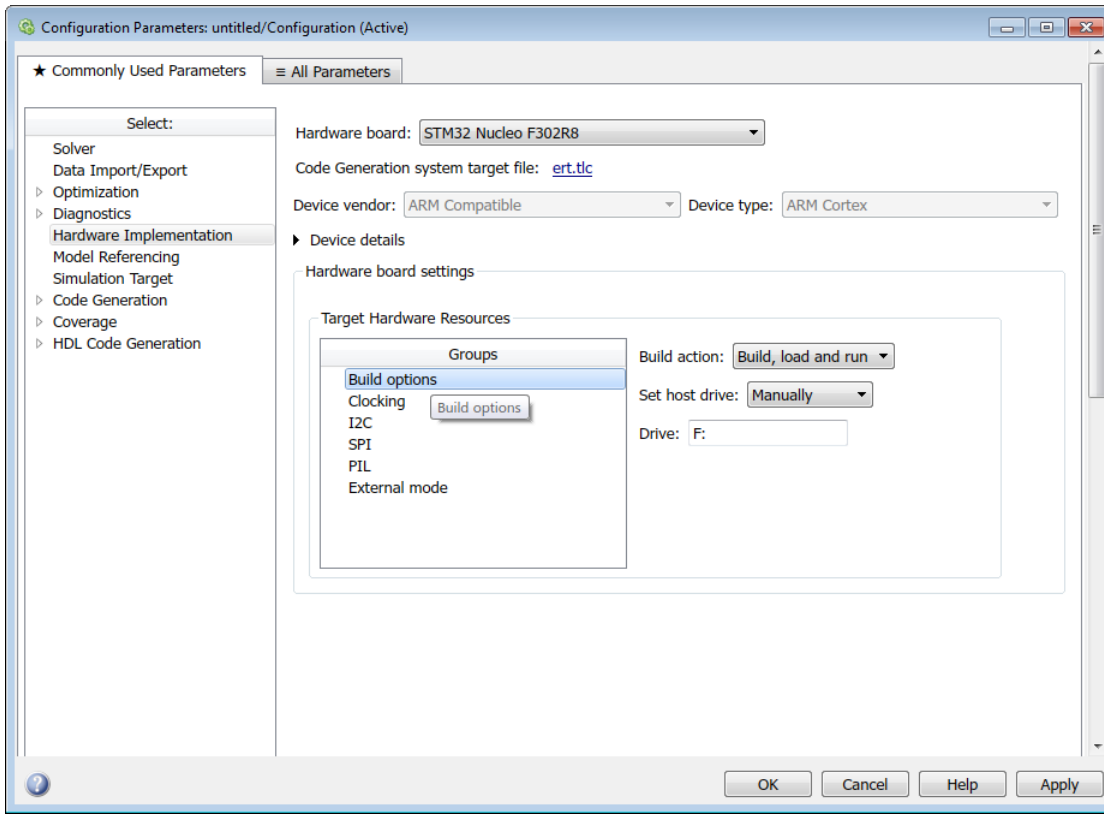


In this section...

Hardware Implementation Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

Build options



Use the build option to specify how the build process should take place during code generation.

Build action

Specify if you want only build or build, load and run actions during code generation.

- `Build` – Select this option if you want to build the code during the build process.
- `Build, load and run` – Select this option to build, load, and to run the generated code during the build process.

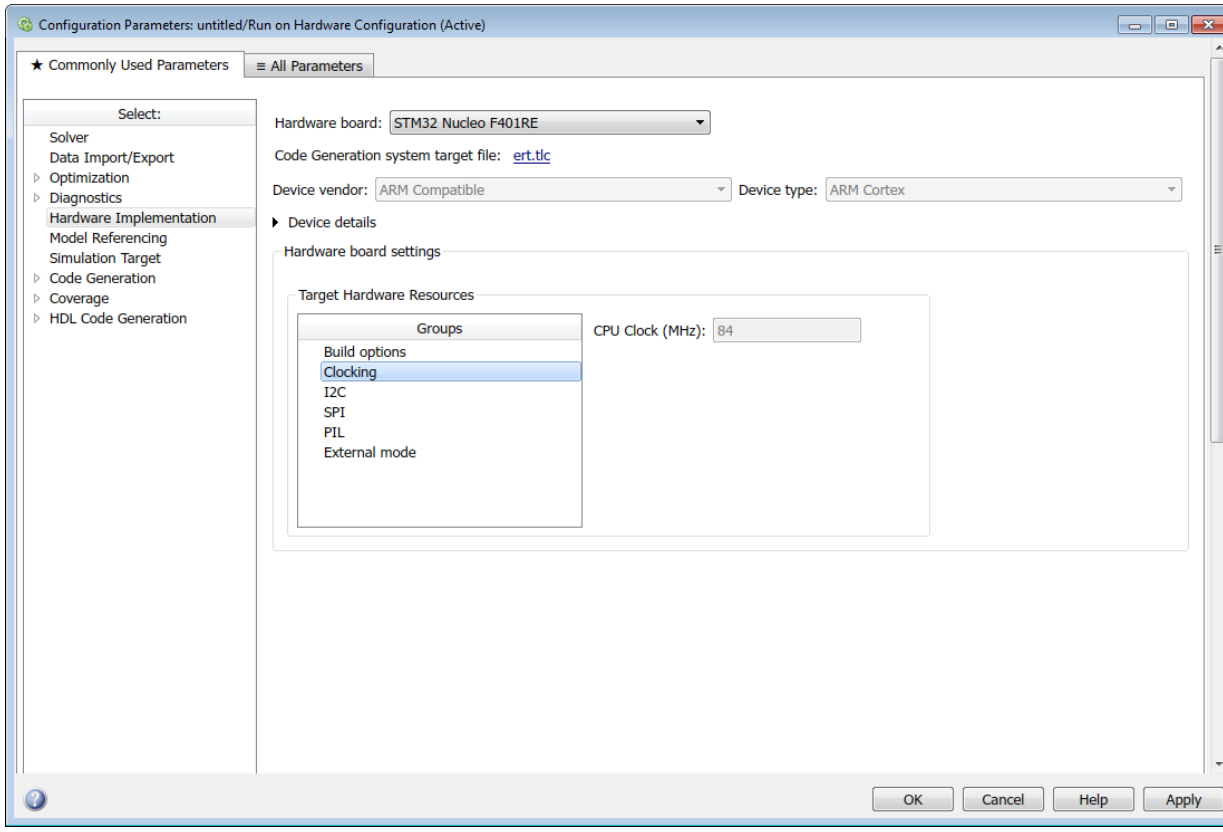
Set host drive

Select an option to copy the generated output bin file automatically or manually on a drive.

Drive

Specify the drive letter on which you want to copy the generated output bin file.

Clocking

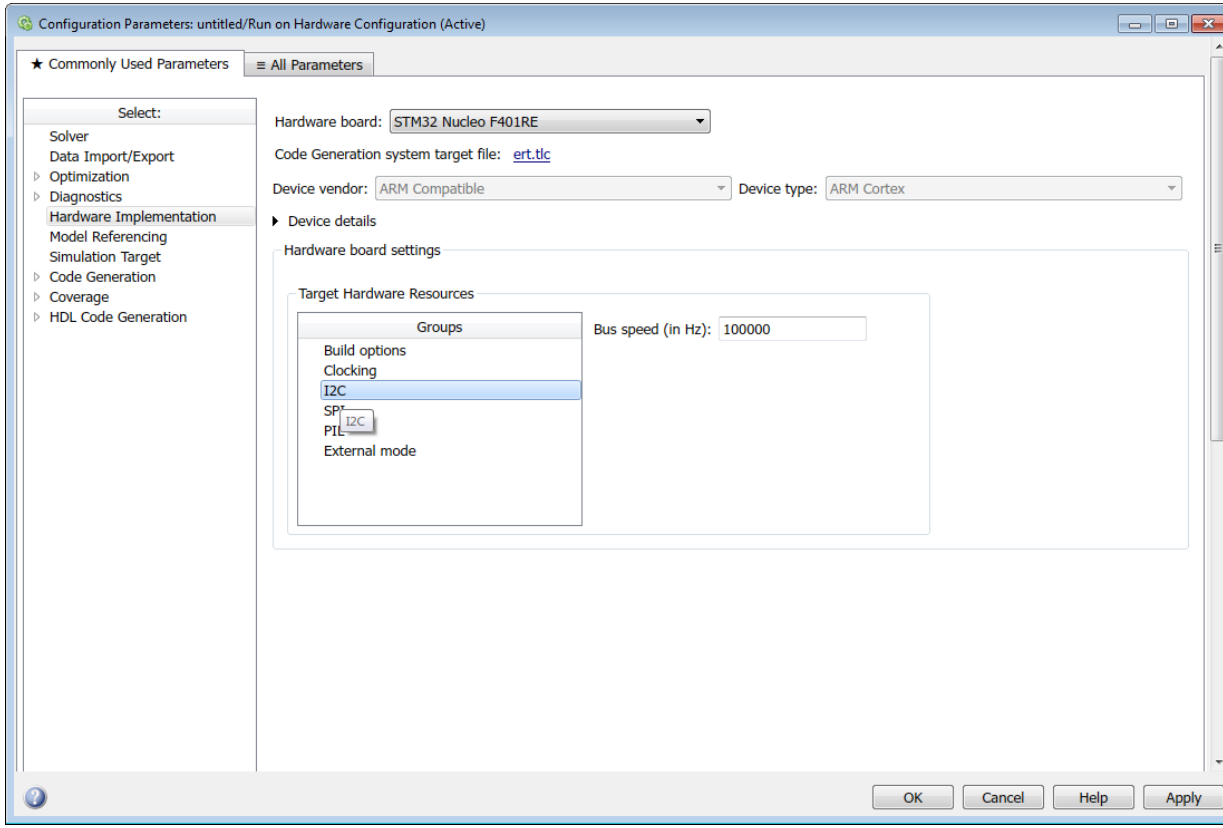


Use the clocking option to achieve the CPU Clock rate specified.

CPU Clock (MHz)

The CPU clock rate.

I2C

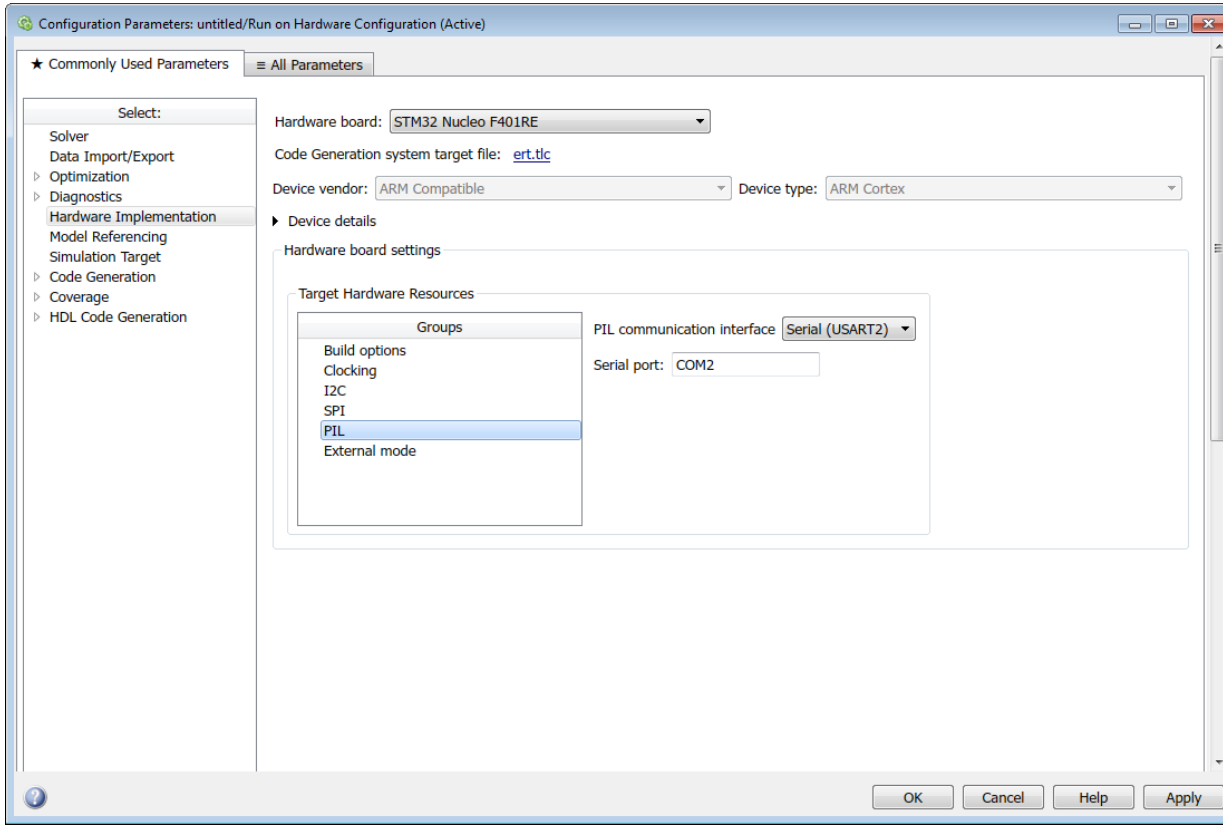


Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

Bus speed (in Hz)

Use the I2C option to set the bus speed parameter. The bus speed determines the rate of data communication between the peripherals that are connected by the I2C bus.

PIL



Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

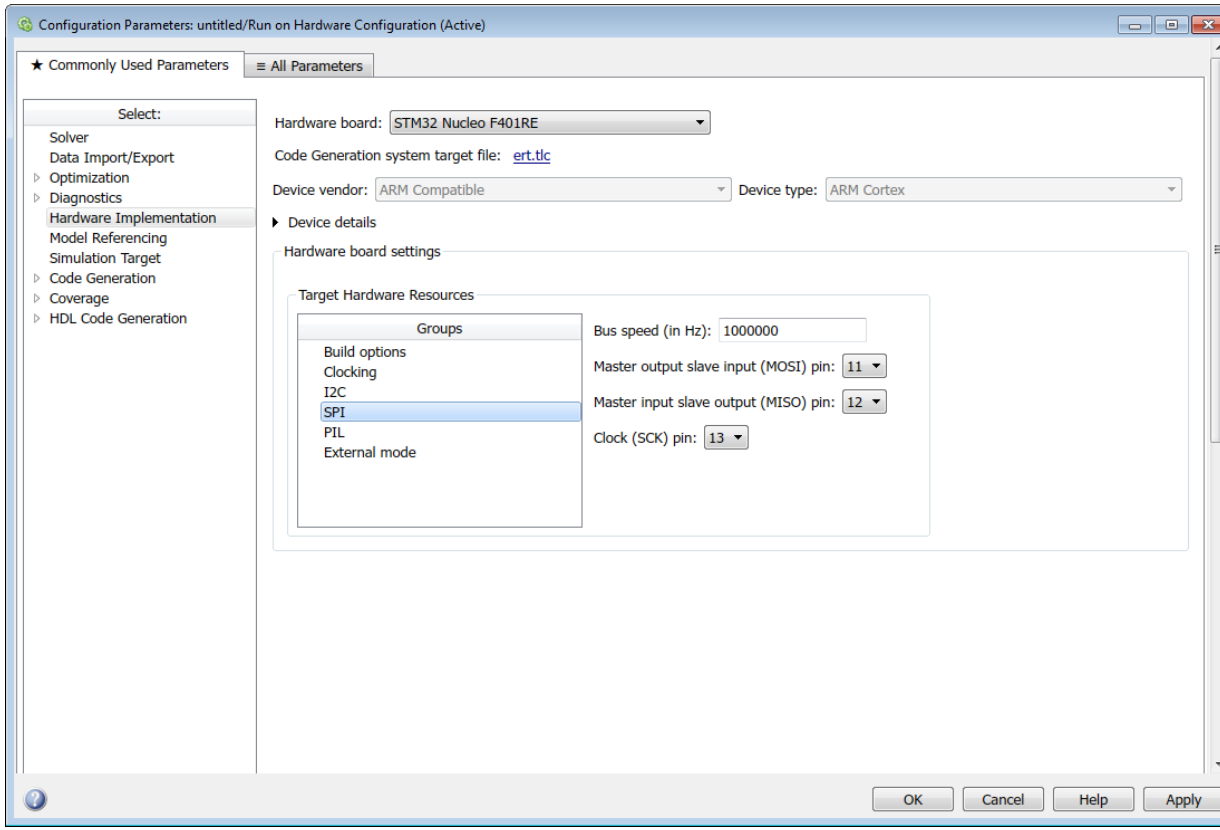
PIL communication interface

The serial port used for PIL communication.

Serial port

Enter the serial port for PIL communication.

SPI



Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

Bus speed (in Hz)

The serial port used for PIL communication.

Master output slave input (MOSI) pin

Specify the pin that connects the master output to the slave input.

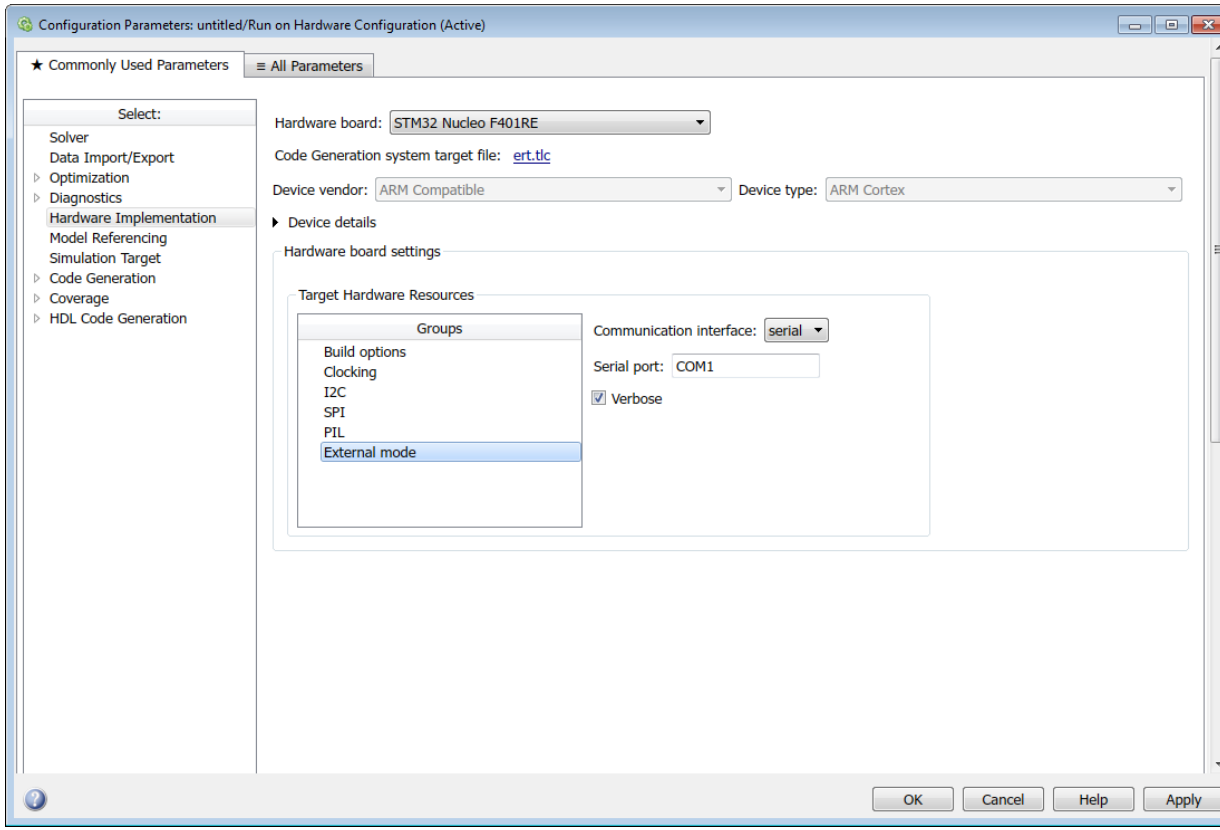
Master input slave input (MISO) pin

Specify the pin that connects the slave output to the master input.

Clock (SCK) pin

Specify the clock pin for SPI communication.

External mode



Communication interface

Use the 'serial' option to run your model in the External mode with serial communication.

Serial port

Enter the serial port used by the target hardware.

Verbose

Select this check box to view the External Mode execution progress and updates in the Diagnostic Viewer or in the MATLAB command window.

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 the "Recommended Settings Summary for Model Configuration Parameters" in the Embedded Coder documentation.

For additional details, click the links in the Configuration Parameter column.

Mapping Application Requirements to the Solver Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Start time	No impact	No impact	No impact	0.0	0.0 seconds
Stop time	No impact	No impact	No impact	A positive value	10.0 seconds
Type	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Variable-step` (you must change to `Fixed-step` for code generation)
"Solver"	No impact	No impact	No impact	`Discrete (no continuous states)`	`ode3 (Bogacki-Shampine)`
"Periodic sample time constraint"	No impact	No impact	No impact	`Specified` or `Ensure sample time independent`	`Unconstrained`
"Sample time properties"	No impact	No impact	No impact	Period, offset, and priority of each sample time	`''`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
				in the model; faster sample times must have higher priority than slower sample times	
"Treat each discrete rate as a separate task"	No impact	No impact	No impact	No impact	On
"Automatically handle rate transition for data transfer"	No impact	No impact (for simulation and during development) Off (for production code generation)	No impact	Off	Off

Mapping Application Requirements to the Data Import/Export Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Input"	No impact	No impact	No recommenda	No recommendati	Off
"Initial state"	No impact	No impact	No recommenda	No recommendati	Off
"Time"	No impact	No impact	No recommenda	No recommendati	On
"States"	No impact	No impact	No recommenda	No recommendati	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Output"	No impact	No impact	No recommenda	No recommendati	On
"Final states"	No impact	No impact	No recommenda	No recommendati	Off
"Signal logging"	No impact	No impact	No recommenda	No recommendati	On
"Record logged workspace data in Simulation Data Inspector"	No impact	No impact	No recommenda	No recommendati	Off
"Limit data points"	No impact	No impact	No recommenda	No recommendati	On
"Decimation"	No impact	No impact	No recommenda	No recommendati	1
"Format"	No impact	No impact	No recommenda	No recommendati	`Array`
"Output options"	No impact	No impact	No recommenda	No recommendati	`Refine output`
"Refine factor"	No impact	No impact	No recommenda	No recommendati	1
"Output times"	No impact	No impact	No recommenda	No recommendati	`'[]'`

Mapping Application Requirements to the Optimization Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Default for underspecified data type"	No impact	No impact	`single`	No impact	`double`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Use division for fixed-point net slope computation"	No impact	No impact	On (when target hardware supports efficient division) Off (otherwise)	No impact	`off`
Application lifespan (days)	No impact	No impact	Finite value	`inf`	`auto`
Use floating-point multiplication to handle net slope corrections	No impact	No impact	On (when target hardware supports efficient multiplicatio Off (otherwise)	No recommendati	Off
Remove code from floating-point to integer conversions that wraps out-of-range values	Off	Off	On (execution, ROM) No impact (RAM)	No recommendati	Off
*The command-line value is reverse of the listed value.					

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Default parameter behavior	`Tunable` (GRT) `Inlined` (ERT)	`Inlined`	`Inlined`	No impact	`Tunable` (GRT) `Inlined` (ERT)
Loop unrolling threshold	No impact	No impact	>0	No impact	5
Maximum stack size (bytes)	No impact	No impact	No impact	No impact	`Inherit from target`
Use memcpy for vector assignment	No impact	No impact	On	No impact	On
Memcpy threshold (bytes)	No impact	No impact	Accept default or determine target-specific optimal value	No impact	64
Inline invariant signals	Off	Off	On	No impact	Off

Mapping Application Requirements to the Optimization Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Use bitsets for storing state configuration"	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Use bitsets for storing Boolean data"	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off

Mapping Application Requirements to the Diagnostics Pane: Solver Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Algebraic loop"	`error`	No impact	No impact	`error`	`warning`
"Minimize algebraic loop"	No impact	No impact	No impact	`error`	`warning`
"Block priority violation"	No impact	No impact	No impact	`error`	`warning`
"Consecutive zero-crossings violation"	No impact	No impact	No impact	`warning` or `error`	`error`
"Unspecified inheritability of sample time"	No impact	No impact	No impact	`error`	`warning`
"Solver data inconsistency"	`warning`	No impact	`none`	No impact	`warning`
"Automatic solver parameter selection"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Sample Time Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Source block specifies -1 sample time"	No impact	No impact	No impact	`error`	`none`
"Multitask rate transition"	No impact	No impact	No impact	`error`	`error`
"Single task rate transition"	No impact	No impact	No impact	`none` or `error`	`none`
"Multitask conditionally executed subsystem"	No impact	No impact	No impact	`error`	`error`
"Tasks with equal priority"	No impact	No impact	No impact	`none` or `error`	`warning`
"Enforce sample times specified by Signal Specification blocks"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Signal resolution"	No impact	No impact	No impact	`Explicit only`	`Explicit only`
"Division by singular matrix"	No impact	No impact	No impact	`error`	`none`
"Underspecified data types"	No impact	No impact	No impact	`error`	`none`
"Simulation range checking"	`warning` or `error`	`warning` or `error`	`none`	`error`	`none`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Wrap on overflow"	No impact	No impact	No impact	`error`	`warning`
"Saturate on overflow"	No impact	No impact	No impact	`error`	`warning`
"Inf or NaN block output"	No impact	No impact	No impact	`error`	`none`
"'rt" prefix for identifiers"	No impact	No impact	No impact	`error`	`error`
"Detect downcast"	No impact	No impact	No impact	`error`	`error`
"Detect overflow"	No impact	No impact	No impact	`error`	`error`
"Detect underflow"	No impact	No impact	No impact	`error`	`none`
"Detect precision loss"	No impact	No impact	No impact	`error`	`error`
"Detect loss of tunability"	No impact	No impact	No impact	`error`	`warning` for GRT-based targets `error` for ERT-based targets
"Detect read before write"	No impact	No impact	No impact	`error`	`Enable all as warnings`
"Detect write after read"	No impact	No impact	No impact	`error`	`Enable all as warning`
"Detect write after write"	No impact	No impact	No impact	`error`	`Enable all as errors`
"Multitask data store"	No impact	No impact	No impact	`error`	`warning`
"Duplicate data store names"	`warning`	No impact	`none`	No impact	`none`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Check undefined subsystem initial output"	No impact	No impact	No impact	On	On
"Check runtime output of execution context"	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"	No impact	No impact	No impact	`warning`	`none`
"Vector/matrix block input conversion"	No impact	No impact	No impact	`error`	`none`
"32-bit integer to single precision float conversion"	No impact	No impact	No impact	`warning`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Connectivity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Signal label mismatch"	No impact	No impact	No impact	`error`	`none`
"Unconnected block input ports"	No impact	No impact	No impact	`error`	`warning`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Unconnected block output ports"	No impact	No impact	No impact	`error`	`warning`
"Unconnected line"	No impact	No impact	No impact	`error`	`none`
"Unspecified bus object at root Outport block"	No impact	No impact	No impact	`error`	`warning`
"Element name mismatch"	No impact	No impact	No impact	`error`	`warning`
"Mux blocks used to create bus signals"	No impact	No impact	No impact	`error`	`error`
"Bus signal treated as vector"	No impact	No impact	No impact	`error`	`warning`
"Invalid function-call connection"	No impact	No impact	No impact	`error`	`error`
"Context-dependent inputs"	No impact	No impact	No impact	`Enable all`	`Use local settings`

Mapping Application Requirements to the Diagnostics Pane: Compatibility Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"S-function upgrades needed"	No impact	No impact	No impact	`error`	`none`

Mapping Application Requirements to the Diagnostics Pane: Model Referencing Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Model block version mismatch"	No impact	No impact	No impact	No recommendati	`none`
"Port and parameter mismatch"	No impact	No impact	No impact	`error`	`none`
"Invalid root Inport/ Outport block connection"	No impact	No impact	No impact	`error`	`none`
"Unsupported data logging"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Saving Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Block diagram contains disabled library links"	No impact	No impact	No impact	No impact	`warning`
"Block diagram contains parameterized library links"	No impact	No impact	No impact	No impact	`none`

Mapping Application Requirements to the Diagnostics Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Unused data, events, messages, and functions"	`warning`	No impact	No impact (for simulation	`warning`	`warning`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
			and during development `none` (for production code generation)		
"Unexpected backtracking"	`warning`	No impact	No impact	`error`	`warning`
"Invalid input data access in chart initialization"	`warning`	No impact	No impact	`error`	`warning`
"No unconditional default transitions"	`warning`	No impact	No impact (for simulation and during development `none` (for production code generation)	`error`	`warning`
"Transition outside natural parent"	`warning`	No impact	No impact (for simulation and during development `none` (for production code generation)	`error`	`warning`

Mapping Application Requirements to the Hardware Implementation Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Hardware board	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`
Device vendor	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	`Intel`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
				using `Custom Processor`.	
Device type	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)`
Number of bits: char	No impact	No impact	Target specific	No recommendati for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type** if they are	`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	
				available in the drop-down list. If your **Device vendor** and **Device type** are not available, set device-specific values by using `Custom Processor`.	
Largest atomic size: integer	No impact	No impact	Target specific	No recommendati 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,	integer `Char`, floating-point `Float`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
				set device-specific values by using `Custom Processor`.	
Byte ordering	No impact	No impact	No impact	No recommendati 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	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 recommendati 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`
Shift right on a signed integer as arithmetic shift	No impact	No impact	On	No recommendati for simulation without code generation.	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
				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`.	
Support long long	No impact	No impact	Target specific	No recommendati for simulation without code generation. For simulation with code generation, select your **Device vendor** and **Device type**	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
				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`.	

Mapping Application Requirements to the Model Referencing Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Rebuild"	No impact	No impact	No impact	`If any changes detected` or `Never` If you use the `Never` setting, then set the **Never rebuild diagnostic** parameter to `Error if rebuild required`	`If any changes detected`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Never rebuild diagnostic"	No impact	No impact	No impact	`error if rebuild required`	`error if rebuild required`
"Enable parallel model reference builds"	No impact	No impact	No impact	No impact	Off
"MATLAB worker initialization for builds"	No impact	No impact	No impact	No impact	`None`
"Total number of instances allowed per top model"	No impact	No impact	No impact	No recommendati	`Multiple`
"Pass fixed-size scalar root inputs by value for code generation"	No impact	No impact	No impact	No recommendati	Off
"Minimize algebraic loop occurrences"	No impact	No impact	No impact	No recommendati	Off
"Propagate sizes of variable-size signals"	No impact	No impact	No impact	No recommendati	`Infer from blocks in model`
"Model dependencies"	No impact	No impact	No impact	No recommendati	`''`

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"	On	No impact	No recommenda	On	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Echo expressions without semicolons"	On	No impact	Off	No impact	On
"Ensure responsiveness"	On	No recommendat	No recommenda	No recommendati	On
"Simulation target build mode"	No impact	No impact	No impact	No impact	`Incremental build`

Mapping Application Requirements to the Simulation Target Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Reserved names"	No impact	No impact	No impact	No recommendati	`{}`

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Parse custom code symbols"	On	No impact	No impact	On	On
"Source file"	No recommendat	No recommendati	No recommend	No recommendatio	`''`
"Header file"	No recommendat	No recommendati	No recommend	No recommendatio	`''`
"Initialize function"	No recommendat	No recommendati	No recommend	No recommendatio	`''`
"Terminate function"	No recommendat	No recommendati	No recommend	No recommendatio	`''`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Include directories"	No impact	No impact	No impact	No recommendatio	''
"Source files"	No impact	No impact	No impact	No recommendatio	''
"Libraries"	No impact	No impact	No impact	No recommendatio	''
"Defines"	No impact	No impact	No impact	No recommendatio	''

Mapping Application Requirements to the Code Generation Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
System target file	No impact	No impact	No impact	No impact (GRT) ERT based (ERT)	`grt.tlc`
Language	No impact	No impact	No impact	No impact	C
Compiler optimization level	`Optimization off (faster builds)`	`Optimizatio off (faster builds)`	`Optimization on (faster runs)` (execution) No impact (ROM, RAM)	No impact	`Optimizations off (faster builds)`
Custom compiler optimization flags	`Optimization off (faster builds)`	`Optimizatio off (faster builds)`	`Optimization on (faster runs)`	No impact	`Optimizations off (faster builds)`
Generate makefile	No impact	No impact	No impact	No impact	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Make command	No impact	No impact	No impact	No recommendati	`make_rtw`
Template makefile	No impact	No impact	No impact	No impact	`grt_default_tmf`
"Select objective" on page 4-28	`Debugging`	Not applicable for GRT-based targets	`Execution efficiency`	No recommendati	`Unspecified`
"Check model before generating code" on page 4-36	`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 4-38	Off	No impact	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Create code generation report" on page 5-4	On	On	No impact	No recommendati	Off
"Open report automatically" on page 5-7	On	On	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Comments Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Include comments	On	On	No impact	No recommendati	On
Simulink block / Stateflow object comments	On	On	No impact	No recommendati	On
Show eliminated blocks	On	On	No impact	No recommendati	On
Verbose comments for Simulink Global storage class	On	On	No impact	No recommendati	On
Operator Annotations	No impact	On	No impact	No recommendati	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	Valid value	>30	No impact	>30	31
Use the same reserved names as Simulation Target	No impact	No impact	No impact	No impact	Off
Reserved names	No impact	No impact	No impact	No impact	{}

Mapping Application Requirements to the Code Generation Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Use the same custom code settings as Simulation Target	No impact	No impact	No impact	No impact	Off
Source file	No impact	No impact	No impact	No impact	' '
Header file	No impact	No impact	No impact	No impact	' '
Initialize function	No impact	No impact	No impact	No impact	' '
Terminate function	No impact	No impact	No impact	No impact	' '
Include directories	No impact	No impact	No impact	No impact	' '
Source files	No impact	No impact	No impact	No impact	' '
Libraries	No impact	No impact	No impact	No impact	' '
Defines	No impact	No impact	No impact	No impact	' '

Mapping Application Requirements (for Debugging) to the All Parameters Tab: Code Generation Category

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Verbose build	On	No impact	No impact	No recommendati	On
Retain .rtw file	On	No impact	No impact	No impact	Off
"Profile TLC" on page 10-65	On	No impact	No impact	No impact	Off
Start TLC debugger when generating code	On	No impact	No impact	No impact	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Start TLC coverage when generating code	On	No impact	No impact	No impact	Off
Enable TLC assertion	On	No impact	No impact	No recommendati	Off

Mapping Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Standard math library	No impact	No impact	Valid library	No impact	`C99 (ISO)`
Code replacement library	No impact	No impact	Valid library	No impact	`None`
Shared code placement	`Shared location` (GRT) No impact (ERT)	`Shared location` (GRT) No impact (ERT)	No impact (execution, RAM) `Shared location` (ROM)	No impact	`Auto`
Support non-finite numbers	No impact	No impact	Off (Execution, ROM) No impact (RAM)	Norecommenda	On
Code interface packaging on page 9-24	No impact	No impact	`Reusable function` or `C++ class`	No impact	`Nonreusable function` if **Language** is set to `C`; `C++ class` if

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
					Language is set to `C++`
Multi-instance code error diagnostic on page 9-28	`Warning` or `Error`	No impact	`None`	No impact	`Error`
Classic call interface on page 10-36	No impact	Off	Off (execution, ROM), No impact (RAM)	No recommendatio	Off (except On for GRT models created before R2012a)
Single output/ update function	On	On	On	No recommendatio	On
MAT-file logging	On	No impact	Off	Off	On (GRT) Off (ERT)
MAT-file variable name modifier	No impact	No impact	No impact	No impact	`rt_`
Generate C API for: signals	No impact	No impact	No impact	No impact (development) Off (production)	Off
Generate C API for: parameters	No impact	No impact	No impact	No impact (development) Off (production)	Off
Generate C API for: states	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	
Generate C API for: root-level I/O	No impact	No impact	No impact	No impact (development) Off (production)	Off
ASAP2 interface	No impact	No impact	No impact	No impact (development) Off (production)	Off
External mode	No impact	No impact	No impact	No impact (development) Off (production)	Off
Transport layer	No impact	No impact	No impact	No impact	`tcpip`
MEX-file arguments	No impact	No impact	No impact	No impact	`''`
Static memory allocation	No impact	No impact	No impact	No impact	Off
Static memory buffer size	No impact	No impact	No impact	No impact	1000000

12

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 Checks”
- “Embedded Coder Checks”

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.
The model or subsystem contains one or more of the following, which require one-based indexing: • Fcn block • MATLAB functions inside Stateflow Charts • MATLAB Function block • MATLAB System block • Stateflow Charts with MATLAB action language • State Transition Table block • Truth Table block	Evaluate the blocks to determine if one-based indexing is used. Consider replacing the blocks with Simulink basic blocks.

Capabilities and Limitations

You can:

- Run this check on your library models.

- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- "cgsl_0101: Zero-based indexing".
- "What Is a Model Advisor Exclusion?"

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** to `Fixed-step` • **Solver** to `Discrete (no continuous states)`
Multitasking diagnostic options are not set to `error`.	In the Configuration Parameters dialog box, on the **Diagnostics** pane, set • **Sample Time** > **Multitask conditionally executed subsystem** to `error` • **Sample Time** > **Multitask rate transition** to `error` • **Data Validity** > **Multitask data store** to `error`

Tips

You do not have to modify the solver settings to generate code from a subsystem. The 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.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- “Supported Products and Block Usage”
- “What Is a Model Advisor Exclusion?”

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 4-11
- "Build configuration" on page 4-13

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	No action.	The model was automatically upgraded.

Condition	Recommended Action	Comment
toolchain approach.		
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`.
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
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

“Hardware Implementation Pane”, “Configure Target 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**

Condition	Recommended Action
	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 Verification and Validation 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?"

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 Referencing Configuration Parameter Requirements

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"** 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"** 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).

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 for each of the code generation objectives.

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precauti (ERT-based targets)	Traceabi (ERT-based targets)	Debuggir (all targets)	MISRA C:2012 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"Check model configuration settings against code generation objectives" on page 12-26	Included	Included	Included	Include	Include	Included	Included	Included
"Check for optimal bus virtuality"	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
"Identify questionable blocks within the specified system" on page 12-25	Included	Included	Included	N/A	N/A	N/A	N/A	N/A
"Check the hardware implementation"	Included if Embedde Coder is available	Included if Embedd Coder	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 precauti (ERT-based targets)	Traceabi (ERT-based targets)	Debuggir (all targets)	MISRA C:2012 guideline (ERT-based targets)	Polyspace (ERT-based targets)
		is available						
"Identify questionable software environment specifications"	Included when Traceabil is not a higher priority and Embedde Coder is available	Included when Traceabi is not a higher priority and Embedd Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
"Identify questionable code instrumentation (data I/O)"	Included when Traceabil or Debuggin are not higher priorities and Embedde Coder is available	Included when Traceabi or Debuggi are not higher priorities and Embedd Coder is available	Included when Traceabi or Debuggi are not higher priorities and Embedd Coder is available	N/A	N/A	N/A	N/A	N/A
"Identify questionable subsystem settings"	N/A	Included if Embedd Coder is available	Included if Embedd 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 precauti (ERT-based targets)	Traceabi (ERT-based targets)	Debuggir (all targets)	MISRA C:2012 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"Identify blocks that generate expensive rounding code"	Included if Embedde Coder is available	Included if Embedd Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
"Identify questionable fixed-point operations"	Included if Embedde Coder or Fixed-Point Designer is available	Included if Embedd Coder or Fixed-Point Designe is available	N/A	N/A	N/A	N/A	N/A	N/A
"Identify blocks using one-based indexing" on page 12-4	Included	Included	N/A	N/A	N/A	N/A	N/A	N/A
"Identify lookup table blocks that generate expensive out-of-range checking code"	Included if Embedde Coder is available	Included if Embedd Coder is available	N/A	N/A	N/A	N/A	N/A	N/A
"Check output types of logic blocks"	Included if Embedde Coder is available	N/A	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 precauti (ERT-based targets)	Traceabi (ERT-based targets)	Debuggir (all targets)	MISRA C:2012 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"Identify unconnected lines, input ports, and output ports"	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A
"Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues"	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A
"Identify block output signals with continuous sample time and non-floating point data type"	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A
"Check for blocks that have constraints on tunable parameters" on page 12-16	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A
"Check if read/write diagnostics are enabled for data store blocks"	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A
"Check structure parameter usage with bus signals"	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A
"Check data store block sample times for modeling errors"	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A
"Check for potential ordering issues involving data store access"	N/A	N/A	N/A	Include	N/A	N/A	N/A	N/A

Check	Execution efficiency (all targets)	ROM efficiency (ERT-based targets)	RAM efficiency (ERT-based targets)	Safety precauti (ERT-based targets)	Traceabi (ERT-based targets)	Debuggir (all targets)	MISRA C:2012 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"Check for blocks not recommended for MISRA C:2012"	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedd Coder is available	N/A

See Also

- "Application Objectives Using Code Generation Advisor" in the Simulink Coder documentation.
- "Configure Model for Code Generation Objectives Using Code Generation Advisor" in the Embedded Coder documentation.
- "Run Model Checks" in the Simulink documentation.
- Simulink Model Advisor Check Reference in the Simulink documentation.
- "Simulink Coder Checks" on page 12-2.
- Simulink Verification and Validation Model Advisor Check Reference in the Simulink Verification and Validation documentation.

Identify questionable blocks within the specified system

Identify blocks not supported by code generation or not recommended for deployment.

Description

The code generator creates code only for the blocks that it supports. Some blocks are not recommended for production code deployment.

Results and Recommended Actions

Condition	Recommended Action
A block is not supported by the 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 Verification and Validation license.

See Also

"Supported Products and Block Usage"

"What Is a Model Advisor Exclusion?"

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.

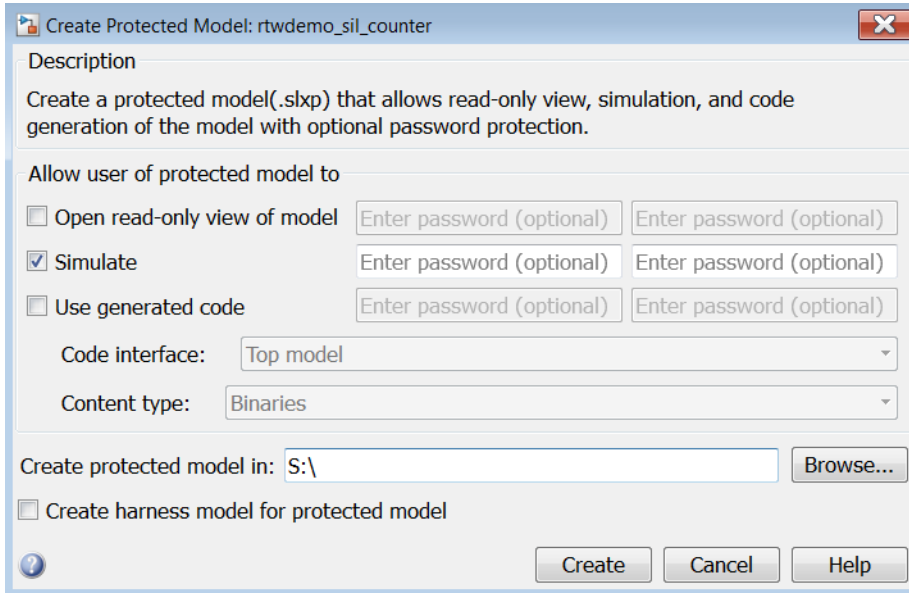
See Also

- "Recommended Settings Summary for Model Configuration Parameters" in the Embedded Coder documentation.
- "Application Objectives Using Code Generation Advisor" in the Simulink Coder documentation.
- "Configure Model for Code Generation Objectives Using Code Generation Advisor" in the Embedded Coder documentation.

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

- "Protected Model"
- "Create a Protected Model"

Open read-only view of model

Share a view-only version of your protected model with optional password protection. View-only version includes the contents and block parameters of the model.

Settings

Default: Off

 On

Share a Web view of the protected model. For password protection, create and verify a password with a minimum of four characters.

 Off

Do not share a Web view of the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Create a Protected Model”
- “Protect a Referenced Model”

Simulate

Allow user to simulate a protected model with optional password protection. Selecting **Simulate**:

- Enables protected model Simulation Report.
- Sets Mode to Accelerator. You can run Normal Mode and Accelerator simulations.
- Displays only binaries and headers.
- Enables code obfuscation.

Settings

Default: On

 On

User can simulate the protected model. For password protection, create and verify a password with a minimum of four characters.

 Off

User cannot simulate the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Create a Protected Model”
- “Protect a Referenced Model”

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 a Protected Model”
- “Protect a Referenced Model”

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 a Protected Model"
- "Protect a Referenced Model"
- "Code Interfaces for SIL and PIL"

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 a Protected Model"
- "Protect a Referenced Model"

Create protected model in

Specify the folder path for the protected model.

Settings

Default: Current working folder

Dependencies

A model that you protect must be available on the MATLAB path.

Alternatives

`Simulink.ModelReference.protect`

See Also

- "Protect a Referenced Model"
- "Create a Protected Model"

Create harness model for protected model

Create a harness model for the protected model. The harness model contains only a Model block that references the protected model.

Settings

Default: Off

 On

Create a harness model for the protected model.

 Off

Do not create a harness model for the protected model.

Alternatives

`Simulink.ModelReference.protect`

See Also

- “Harness Model”
- “Test the Protected Model”

Tools — Alphabetical List

Code Replacement Viewer

Explore content of code replacement libraries

Description

The Code Replacement Viewer displays the content of code replacement libraries. Use the tool to explore and choose a library. If you develop a custom code replacement library, use the tool to verify table entries.

- 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 that the library contains. When you select a code replacement table, the viewer displays function and operator code replacement entries that are in that table.

Abbreviated Entry Information

In the middle pane, the viewer displays entries that are in the selected code replacement table, along with abbreviated information for each entry.

Field	Description
Name	Name or identifier of the function or operator being replaced (for example, `cos` or `RTW_OP_ADD`).
Implementation	Name of the implementation function, which can match or differ from **Name**.
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.

Field	Description
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`

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`

Field	Description
	`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

Field	Description
	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.

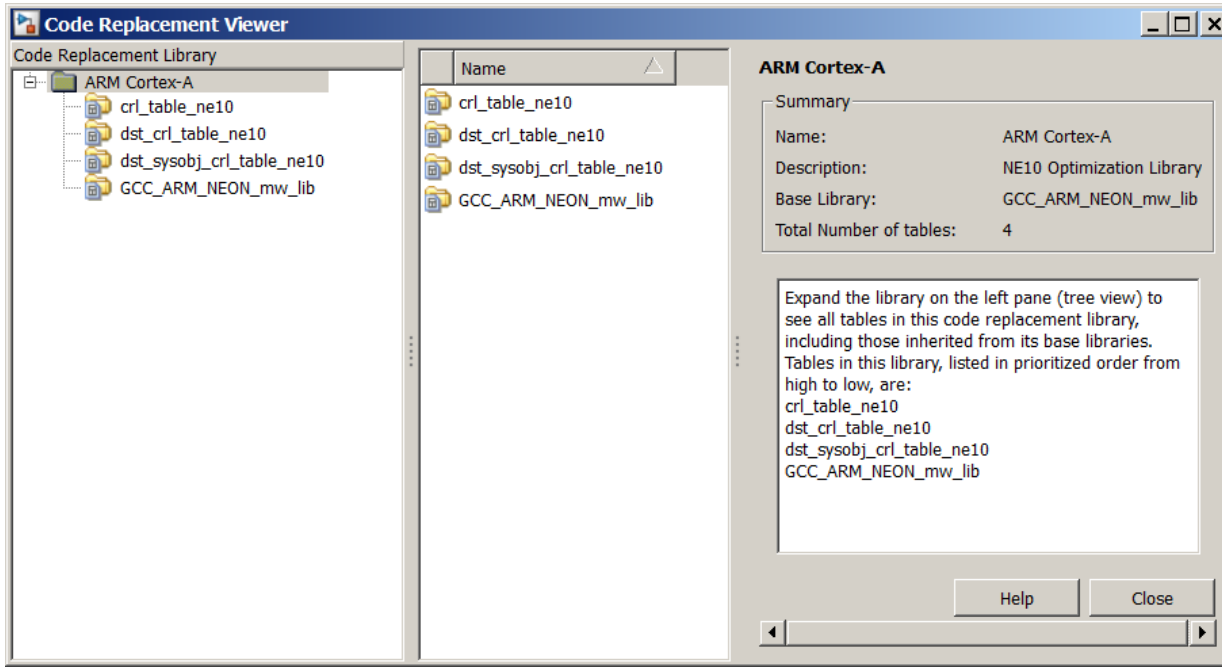
Open the Code Replacement Viewer App

Open from the MATLAB command prompt using `crviewer`.

Examples

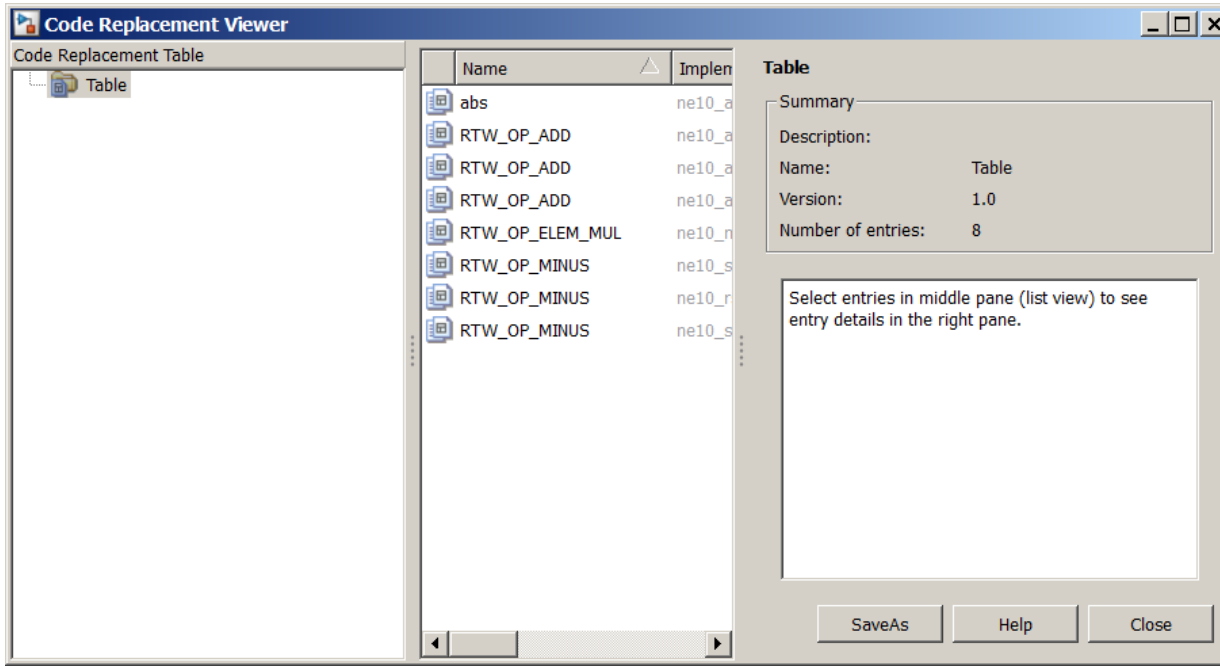
Display Contents of Code Replacement Library

```
crviewer('ARM Cortex-A')
```



Display Contents of Code Replacement Table

```
crviewer(clr_table_ne10)
```



- "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. For example, `'GNU 99 extensions'`.

`crviewer(table)` opens the Code Replacement Viewer and displays the contents of `table`, where `table` is a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

More About

- "What Is Code Replacement?"
- "Code Replacement Libraries"
- "Code Replacement Terminology"