Simulink® Coder™

Reference

R2014**b**
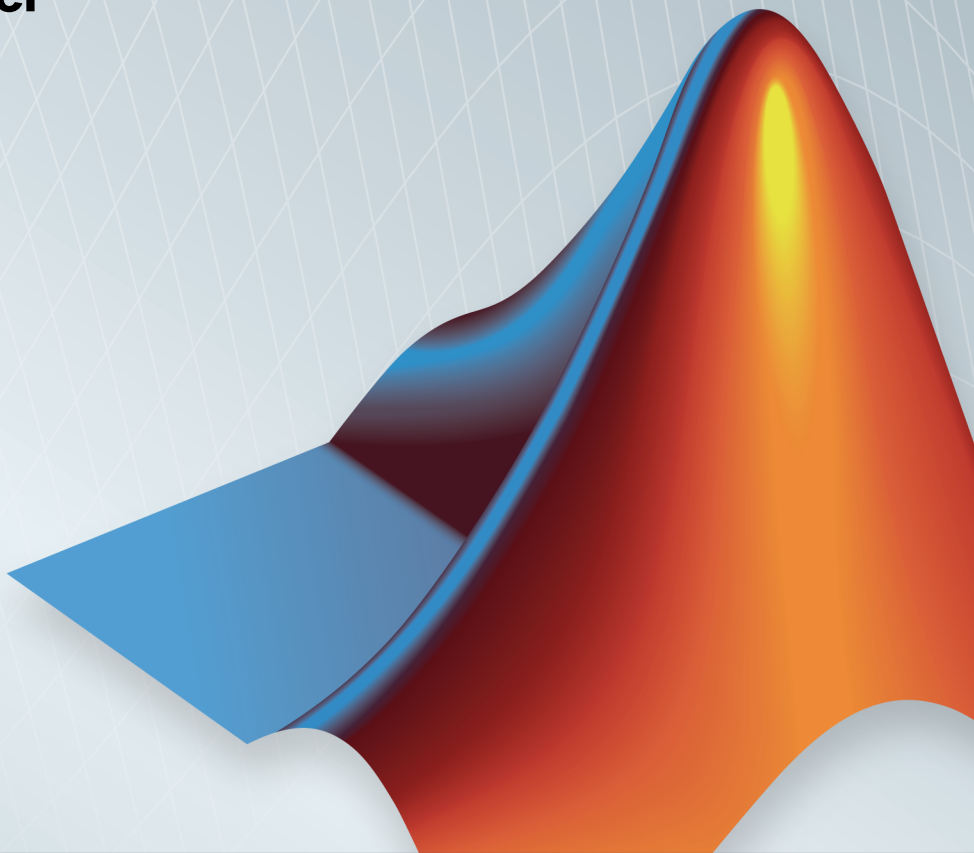
MATLAB®&SIMULINK®
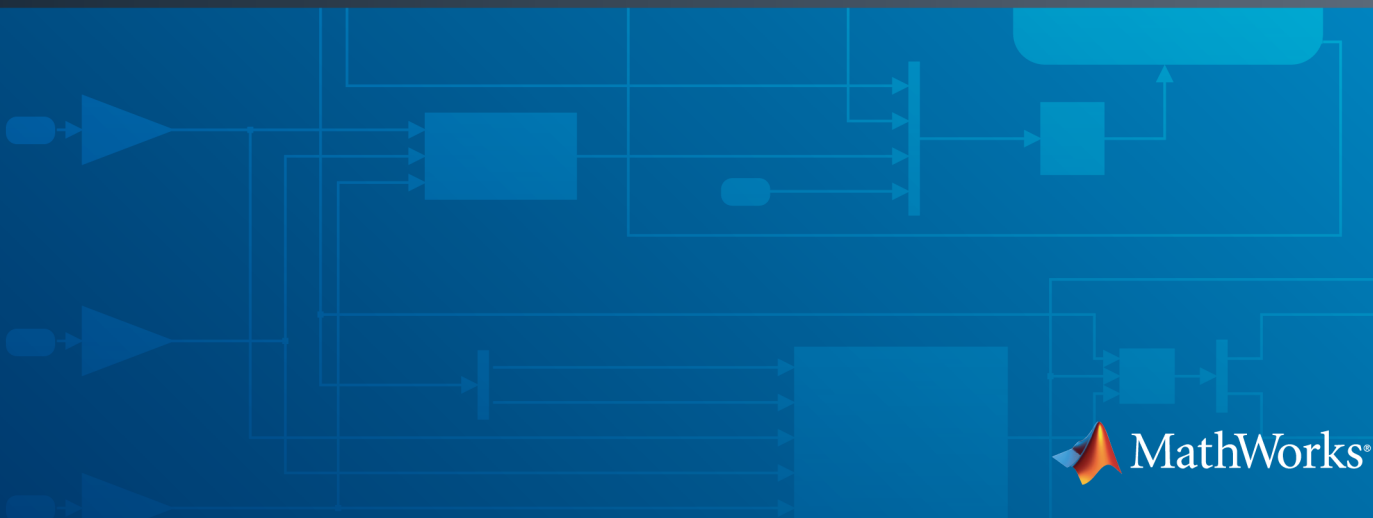
MathWorks®

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

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

Model Advisor Checks

5

Parameters for Creating Protected Models

6

Tools — Alphabetical List

7

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"
- "Limitations on Data Type Specifications in Workspace"
- "Code Reuse Limitations for Subsystems"
- "Simulink Coder Model Referencing Limitations"
- "External Mode Limitations"
- "Noninlined S-Function Parameter Type Limitations"
- "S-Function Target Limitations"
- "Rapid Simulation Target Limitations"
- "Asynchronous Support Limitations"
- "C API Limitations"
- "Supported Products and Block Usage"

Alphabetical List

addCompileFlags

Add compiler options to model build information

Syntax

addCompileFlags(*buildinfo, options, groups*)

groups is optional.

Arguments

buildinfo

Build information returned by RTW.BuildInfo.

options

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

'-Zi -Wall'   '-O3'

groups (optional)

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

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

You can apply

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

To...	Specify *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 Simulink Coder™ makefile build process, specify *groups* as either `'OPTS'` or `'OPT_OPTS'`.

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

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

Description

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

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

Examples

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

addDefines | addLinkFlags | getCompileFlags

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 string to the vector as a single element. For example, if you add `'-DPROTO -DDEBUG'` and then `'-DPRODUCTION'`, the vector consists of two elements, as shown below.

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

groups (optional)

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

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

You can apply

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

To...	Specify *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 Simulink Coder makefile build process, specify *groups* as either `'OPTS'` or `'OPT_OPTS'`.

Description

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

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

Examples

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

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

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

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

- 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

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

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

The function removes duplicate include file entries that

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

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

paths (optional)

A character array or cell array of character arrays that specifies paths to the include files. The function adds the paths to the end of a vector in the order that you specify

them. If you specify a single path as a character array, the function uses that path for all files.

groups (optional)

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

- 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 Simulink Coder software stores the include files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

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

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

If You Specify an Optional Argument as a...	The Function...
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 string (' ') for *paths*.

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

Examples

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

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

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

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

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

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

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

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

More About

• "Customize Post-Code-Generation Build Processing"

See Also

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

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 string and corresponding filename.

groups (optional)

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

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

You can apply

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

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 Simulink Coder software stores the include paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

addIncludeFiles | addSourceFiles | addSourcePaths | getIncludePaths |
updateFilePathsAndExtensions | updateFileSeparator

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 Simulink Coder makefile build process, specify *groups* as either `'OPTS'` or `'OPT_OPTS'`.

Description

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

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

Examples

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

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

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

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

- 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

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 string and corresponding linkable object filename.

paths

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

priority (optional)

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

precompiled (optional)

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

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

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

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

linkonly (optional)

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

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

If *linkonly* is `false` (the default), rules for building the link objects are added to the makefile. In this case, the value of *precompiled* determines which subsection of the added rules is expanded, `START_PRECOMP_LIBRARIES` (`true`) or `START_EXPAND_LIBRARIES` (`false`). 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 Simulink Coder software stores the link objects in a vector in relative priority order. If multiple objects have the same priority or you do not specify priorities, the function adds the objects to the vector based on the order in which you specify them.

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

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"

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

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

The function removes duplicate nonbuild file entries that

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

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

paths (optional)

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

groups (optional)

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

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

You can apply

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

To...	Specify *groups* as a...
Apply one group name to one or more nonbuild files	Character array.
Apply different group names to nonbuild files	Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for *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 Simulink Coder software stores the nonbuild files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

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

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 string (' ') for *paths*.

Examples

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also
getNonBuildFiles

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

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

The function removes duplicate source file entries that

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

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

paths (optional)

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

groups (optional)

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

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

You can apply

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

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

Description

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

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

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

2-27

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

Examples

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

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

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

addIncludeFiles | addIncludePaths | addSourcePaths | getSourceFiles |
updateFilePathsAndExtensions | updateFileSeparator

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 string and corresponding filename.

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

---

groups (optional)

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

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

You can apply

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

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

Description

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

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

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

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

Examples

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

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

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

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

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

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also

addIncludeFiles | addIncludePaths | addSourceFiles | getSourcePaths | updateFilePathsAndExtensions | updateFileSeparator

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 2-34.)

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

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

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

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`

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
string

Model name specified as a string

Example: 'rtwdemo_counter'

Data Types: char

`subsystem` — Subsystem name
string

Subsystem name specified as a string

Example: 'rtwdemo_counter/Amplifier'

Data Types: char

Name-Value Pair Arguments

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

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

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

Navigation

`'IncludeHyperlinkInReport'` — Code-to-model hyperlinks
'off' | 'on'

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

Example: `'IncludeHyperlinkInReport','on'`

Data Types: `char`

`'GenerateTraceInfo'` — Model-to-code highlighting
'off' | 'on'

Model-to-code highlighting, specified as 'on' or 'off'. Specify 'on' to include model-to-code highlighting in the code generation report. For more information see "Model-to-code".

Example: `'GenerateTraceInfo','on'`

Data Types: `char`

`'GenerateWebview'` — Model Web view
'off' | 'on'

Model Web view, specified as 'on' or 'off'. Specify 'on' to include the model Web view in the code generation report. For more information, see "Generate model Web view".

Example: `'GenerateWebview','on'`

Data Types: `char`

Traceability Report Contents

`'GenerateTraceReport'` — Summary of eliminated and virtual blocks
'off' | 'on'

Summary of eliminated and virtual blocks, specified as 'on' or 'off'. Specify 'on' to include a summary of eliminated and virtual blocks in the code generation report. For more information, see "Eliminated / virtual blocks".

Example: ''GenerateTraceReport','on''

Data Types: char

`'GenerateTraceReportSl'` — Summary of Simulink blocks and the corresponding code location
'off' | 'on'

Summary of the Simulink blocks and the corresponding code location, specified as 'on' or 'off'. Specify 'on' to include a summary of the Simulink blocks and the corresponding code location in the code generation report. For more information, see "Traceable Simulink blocks".

Example: ''GenerateTraceReportSl','on''

Data Types: char

`'GenerateTraceReportsSf'` — Summary of Stateflow objects and the corresponding code location
'off' | 'on'

Summary of the Stateflow objects and the corresponding code location, specified as 'on' or 'off'. Specify 'on' to include a summary of Stateflow objects and the corresponding code location in the code generation report. For more information, see "Traceable Stateflow objects".

Example: ''GenerateTraceReportSf','on''

Data Types: char

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

Summary of the MATLAB functions and the corresponding code location, specified as 'on' or 'off'. Specify 'on' to include a summary of the MATLAB objects and the corresponding code location in the code generation report. For more information, see "Traceable MATLAB functions".

Example: ''GenerateTraceReportEml','on''

Data Types: char

Metrics

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

Static code metrics, specified as 'on' or 'off'. Specify 'on' to include static code metrics in the code generation report. For more information, see "Static code metrics".

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

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
string

Model name specified as a string

Example: 'rtwdemo_counter'

Data Types: char

subsystem — Subsystem name
string

Subsystem name specified as a string

Example: 'rtwdemo_counter/Amplifier'

Data Types: char

More About

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

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

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.

More About

• "Customize Post-Code-Generation Build Processing"

See Also

getBuildArgs

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 `*`.
- 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

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 string (`''`) for *includeGroupIDs*.

More About

· "Customize Post-Code-Generation Build Processing"

See Also

`findBuildArg`

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 string (' ') 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

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 string (' ')

Description

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

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

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

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

Examples

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

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

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

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

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

Output Arguments

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

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

---

updateFilePathsAndExtensions to resolve file paths and extensions before calling getFullFileList.

Description

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

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

Examples

List the files stored in build information myModelBuildInfo.

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

More About

- "Customize Post-Code-Generation Build Processing"

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 string for your MATLAB installation folder.

getIncludeFiles

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 string ('') for *includeGroups*.

Examples

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

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

2-59

```
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
```

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 string for your MATLAB installation folder.
false	Does not replace the token $(MATLAB_ROOT).

includeGroups (optional)

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

excludeGroups (optional)

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

Output Arguments

Paths of include files stored in the model build information.

Description

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

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

Examples

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

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo, {'/etc/proj/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

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 string (' ') 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

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 string 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 string (`' '`) for *includeGroups*.

Examples

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

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

nonbuildfiles =
```

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

More About

- "Customize Post-Code-Generation Build Processing"

See Also
addNonBuildFiles

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 string.

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

includeGroups (optional)

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

excludeGroups (optional)

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

Output Arguments

Names of source files stored in the model build information.

Description

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

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

Examples

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

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

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 string for your MATLAB installation folder.
false	Does not replace the token $(MATLAB_ROOT).

includeGroups (optional)

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

excludeGroups (optional)

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

Output Arguments

Paths of source files stored in the model build information.

Description

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

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

Examples

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

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

srcpaths =

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

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

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

srcpaths =

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

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

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

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

ans =

$(MATLAB_ROOT)\rtw\c\src

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

ans =

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

More About

• "Customize Post-Code-Generation Build Processing"

See Also

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

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**":

- `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 "Function Prototype Control" 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

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 "**Tasking mode for periodic sample times**" (SolverMode) parameter specified for the model:

Tasking Mode	Function Prototype
SingleTasking (single-rate or multirate)	void *model*_step(void);
MultiTasking (multirate)	void *model*_step*N* (void); (*N* is a task identifier)

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

- C++ class (default for C++ language) — Generated function is encapsulated into a C ++ class method. Required model data is encapsulated into C++ class attributes.

- Nonreusable function (default for C language) — Generated function passes (void). Model data structures are statically allocated, global, and accessed directly in the model code.

- Reusable function — Generated function passes the real-time model data structure, by reference, as an input argument. The real-time model data structure is exported with the *model*.h header file.

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

> **Note:** If you have an Embedded Coder license:
>
> • For `Nonreusable function` code interface packaging, you can use the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters
>
>   dialog box. For more information, see "Function Prototype Control" in the Embedded Coder documentation.
> • For `C++ class` code interface packaging, you can use the **Configure C++ Class Interface** button and related controls on the **Interface** pane of the Configuration Parameters dialog box. For more information, see "C++ Class Interface Control" in the Embedded Coder documentation.

Description

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

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

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

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

For 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

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**":

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

More About

· "Entry-Point Functions and Scheduling"

See Also

model_initialize | model_step

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'`	`'string'` 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**" 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"

rsimgetrtp

Global model parameter structure

Syntax

```
parameter_structure = rsimgetrtp('model')
parameter_structure =
rsimgetrtp('model','AddTunableParamInfo','value')
```

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.

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

The model parameter structure contains the following fields:

Field	Description
modelChecksum	A four-element vector that encodes the structure. The Simulink Coder software 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
complex	Value 1 if parameter values are complex and 0 if real
dtTransIdx	Internal use only
values	Vector of parameter values

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

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

Examples

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

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

param_struct =

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

More About

- "Create a MAT-File That Includes a Model Parameter Structure"
- "Update a Block Diagram"
- "Inline parameters"

- "Block Creation"
- "Tune Parameters"

See Also

`rsimsetrtpparam`

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 `idx`th parameter set. There can be more than one name-value pair. If the `rtP` structure does not have `idx` parameter sets, the first set is copied and appended until there are `idx` parameter sets. Subsequently, the `idx`th set is changed.

Input Arguments

rtP

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

idx

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

paramValue

The value of the `rtP` parameter `paramName`.

paramName

The name of the parameter set to add to the `rtP` structure.

Output Arguments

rtP

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

Definitions

The `rtP` structure should match the format of the structure returned by `rsimsetrtp(modelName)`.

Examples

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

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

2   Add three parameter sets to the parameter structure `rtp`.

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

See Also

rsimgetrtp

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 strings 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 blocks that use precompiled libraries that the rtwmakecfg files specify. The template

Field	Value
	makefile (TMF)-to-makefile conversion generates the library rules only if the conversion needs the libraries.
libSuffix (optional)	String that specifies the suffix, including the file type extension, to append to the name of each library (for example, .a or _vc.lib). The string 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)	String 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 TMF can specify other libraries and how to build them. Use this field if you must precompile libraries.

Examples

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

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

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

% 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

- "Precompile S-Function Libraries"
- "Recompile Precompiled Libraries"

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.

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

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

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

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

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

```
rtwbuild(subsystem,'Mode','ExportFunctionCalls,
'ExportFunctionInitializeFunctionName', fcnname) names the exported
```
initialization function, specified as a string, 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 target, the coder generates 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 rtwtypes.h multiword_types. builtin_typeid_t	rtmodel.l	rt_logging.c

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

Parameter Setting	Results
Code Generation > **Generate code only** pane is cleared	Executable image rtwdemo_rtwintro.exe

Parameter Setting	Results
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 Amplifiler in model rtwdemo_rtwintro.

```
rtwbuild('rtwdemo_rtwintro/Amplifier')
```

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

Model Files	Shared Files	Interface Files	Other Files
Amplifier.c Amplifier.h Amplifier_priva Aplifier_types.	rtGetInf.c rtGetInf.h rtGetNaN.c rtGetNaN.h rt_nonfinite.c rt_nonfinite.h rtwtypes.h	rtmodel.h	rt_logging.c

Model Files	Shared Files	Interface Files	Other Files
	`multiword_types.h` `builtin_typeid_typ`		

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

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

Build Subsystem for Exporting Code to External Application

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

```
rtwdemo_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 **Code Generation** > **Verification** > **Create block** to SIL.

Create the SIL block.

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

The coder generates a SIL block for the generated subsystem code. You can add the block to an environment or test harness model that supplies test vectors or stimulus input. You

can then run simulations that perform SIL tests and verify that the generated code in the SIL block produces the same result as the original subsystem.

Name Exported Initialization Function

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

```
rtwdemo_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 string 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 string 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 coder to regenerate code for the top model of a system that includes referenced models	`true`
Let the coder determine whether to regenerate top model code based on model and model parameter changes	`false`

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

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

Alternatively, you can force regeneration of top model code by deleting code generation folders, such as `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`
Display build error messages in the Command Window only	`false`

Output Arguments

`blockHandle` — Handle to SIL block created for generated subsystem code
handle

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

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

More About

Tips

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

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

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

See Also
slbuild

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

model1 — Input data
character string

Character string specifying the name of a Simulink model.

Example: 'sldemo_fuelsys'

Data Types: char

Output Arguments

`folderstruct` — Output data
structure

Structure containing the following:

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

More About

- "Working Folder"

- "Folders Used During the Build Process"
- "Control the Location for Generated Files"

See Also

```
rtwbuild
```

rtwrebuild

Rebuild generated code

Syntax

```
rtwrebuild()
rtwrebuild('model')
rtwrebuild('path')
```

Description

`rtwrebuild()` recompiles generated code files you modified by invoking the makefile generated during the previous build. If you omit the input arguments, the current working folder must be the build folder of the model (not the model location).

Use `rtwrebuild('model')` if your current working folder is one level above the build folder of the model (`pwd` when you initiated the Simulink Coder build).

Use `rtwrebuild('path')` to specify the path to the build folder of the model.

If your model includes referenced models, the Simulink Coder software builds the referenced models recursively before rebuilding the top model.

Input Arguments

model	String specifying the model name.
path	String specifying the fully qualified path to the build folder for the model.

Examples

Rebuild generated code for a model located in the current working folder (one level above its build folder):

```
rtwrebuild('mymodel')
```

Rebuild generated code for a model by specifying a path to its build folder:

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

More About

- "Rebuild a Model"

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 Simulink Coder software names the report codegen.html. It places the file in your current folder. The report includes:

- Snapshots of the model, including subsystems.
- Block execution order list.
- Code generation summary with a list of generated code files, configuration settings, a subsystem map, and a traceability report.
- Full listings of generated code that reside in the build folder.

rtwreport(model, folder) specifies the build folder, *model_target*_rtw. The Simulink code generation folder, slprj, must reside in the parent folder of folder. If the software cannot find the folder, an error occurs and code is not generated.

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
string

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

Example: `'rtwdemo_secondOrderSystem'`

Data Types: `char`

`folder` — Build folder name
string

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

Example: `'rtwdemo_secondOrderSystem_grt_rtw'`

Data Types: `char`

More About

- "Report Explorer"
- "Code Generation Summary"

Related Examples

- "Document Generated Code with Simulink Report Generator"
- "Import Generated Code"

rtwtrace

Trace a block to generated code in HTML code generation report

Syntax

```
rtwtrace('blockpath')
```

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 **Code Generation** > **Report** pane, and select the **Model-to-code** parameter.
- You generate code for the model using the Embedded Coder software.
- You have the build folder under the current working folder; otherwise, `rtwtrace` may produce an error.

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')
```
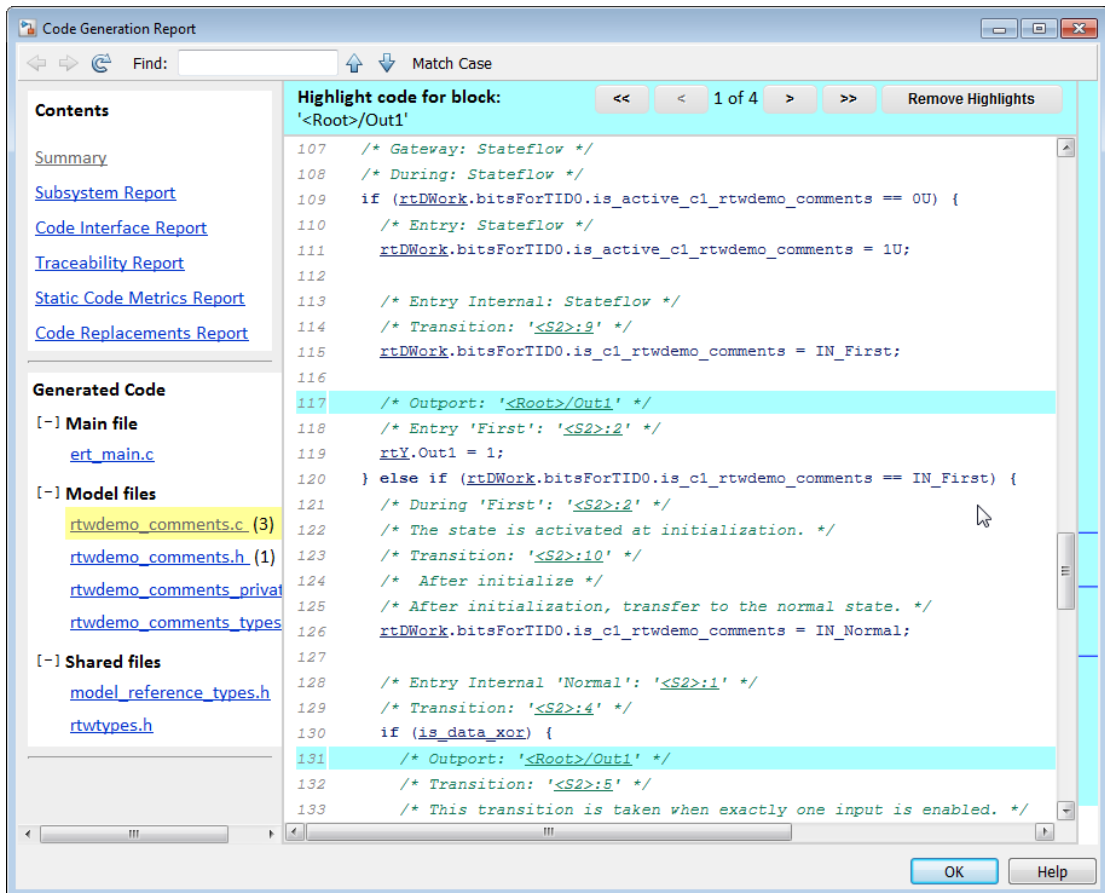
Input Arguments

`blockpath` — block path
string

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

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**.

Related Examples

- "Trace Model Objects to Generated Code"
- "Model-to-code"

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 Simulink Coder code generation files. The initial session defaults for these parameters are provided by the Simulink preferences "Simulation cache folder" and "Code generation folder".

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

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

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

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

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

Input Arguments

`action`

String specifying one of the following actions:

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

`'CodeGenFolder', codeGenFolderPath`

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

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

- `'C:\Work\mymodelsimcache'` and `'/mywork/mymodelgencode'` specify absolute paths.

- `'mymodelsimcache'` is a path relative to the current working folder (`pwd`). The software converts a relative path to a fully qualified path at the time the `CacheFolder` or `CodeGenFolder` parameter is set. For example, if `pwd` is `'/mywork'`, the result is `'/mywork/mymodelsimcache'`.

- `'../test/mymodelgencode'` is a path relative to `pwd`. If `pwd` is `'/mywork'`, the result is `'/test/mymodelgencode'`.

`'keepPreviousPath', true`

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

`'createDir', true`

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

Output Arguments

cfg

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

Examples

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

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

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

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

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

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

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

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

Alternatives

Instead of setting the `CacheFolder` and `CodeGenFolder` parameters just for the current MATLAB session, you can set the Simulink preferences "Simulation cache folder"

and "Code generation folder", which provide the initial MATLAB session defaults. The preferences can be set using the Simulink Preferences dialog box or using the MATLAB command `set_param`.

More About

- "Control the Location for Generated Files"

See Also

"Code generation folder" | "Simulation cache folder"

Simulink.ModelReference.modifyProtectedModel

Modify existing protected model

Syntax

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

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

Description

`Simulink.ModelReference.modifyProtectedModel(model)` modifies options for an existing protected model created from the specified model. If Name,Value pair arguments are not specified, the modified protected model is updated with default values and supports only simulation.

`Simulink.ModelReference.modifyProtectedModel(model,Name,Value)` uses additional options specified by one or more Name,Value pair arguments. These options are the same options that are provided by the `Simulink.ModelReference.protect` function. However, these options have additional options to change encryption passwords for read-only view, simulation, and code generation. When you add functionality to the protected model or change encryption passwords, the unprotected model must be available. The software searches for the model on the MATLAB path. If the model is not found, the software reports an error.

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

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

Examples

Update Protected Model with Default Values

Create a modifiable protected model with support for code generation, then reset it to default values.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Modify the model to use default values.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter');
```

The resulting protected model is updated with default values and supports only simulation.

Remove Functionality from Protected Model

Create a modifiable protected model with support for code generation and Web view, then modify it to remove the Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Webview',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Remove support for Web view from the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter', 'Mode', 'CodeGeneration','Report',true);
```

Change Encryption Password for Code Generation

Change an encryption password for a modifiable protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add the password that the protected model user must provide to generate code.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','cgpassword');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Encrypt',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Change the encryption password for simulation.

```
Simulink.ModelReference.modifyProtectedModel(
'sldemo_mdlref_counter','Mode','CodeGeneration','Encrypt',true,...
'Report',true,'ChangeSimulationPassword',...
{'cgpassword','new_password'});
```

Add Harness Model for Protected Model

Add a harness model for an existing protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add a harness model for the protected model.

```
[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Report',true,...
'Harness',true);
```

Input Arguments

`model` — Model name
string (default)

Model name, specified as a string. It contains the name of a model or the path name of a Model block that references the 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) | string

Folder for protected model, specified as a string.

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

`'Report'` — Option to generate a report
false (default) | true

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

To view the report, double-click the protected model block. The report is generated in HTML format. It includes 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 string. Specify one of the following values:

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

Example: `'Mode','Accelerator'`

`'OutputFormat'` — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

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

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

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: All code in the build 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.

Example: `'Webview',true`

Encryption

`'ChangeSimulationPassword'` — Option to change the encryption password for simulation
cell array of two strings

Option to change the encryption password for simulation, specified as a cell array of two strings. The first string is the old password, the second string 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 strings

Option to change the encryption password for read-only view, specified as a cell array of two strings. The first string is the old password, the second string is the new password.

Example: `'ChangeViewPassword',{'old_password','new_password'}`

`'ChangeCodeGenerationPassword'` — Option to change the encryption password for code generation
cell array of two strings

Option to change the encryption password for code generation, specified as a cell array of two strings. The first string is the old password, the second string 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`

2-125

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');
```

2-127

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"

Input Arguments

`model` — Model name
string (default)

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

Name-Value Pair Arguments

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

Example:
`'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that

only binary files and headers in the generated code be visible to users of the protected model.

`'Harness'` — Option to create a harness model
false (default) | true

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

Example: `'Harness',true`

`'Mode'` — Model protection mode
`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'ViewOnly'`

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

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

Example: `'Mode','Accelerator'`

`'ObfuscateCode'` — Option to obfuscate generated code
true (default) | false

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

Example: `'ObfuscateCode',true`

`'Path'` — Folder for protected model
current working folder (default) | string

Folder for protected model, specified as a string.

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

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: All code in the build 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.

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`

2-131

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"

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
```

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
```

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
string

Model name specified as a string

Example: 'rtwdemo_counter'

Data Types: char

More About

·   "Protect a Referenced Model"

See Also

`Simulink.ModelReference.ProtectedModel.clearPasswords`

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 strings

A list of exported symbols generated by protected model that you must not modify. Default value is empty.

`NonSourceFiles` — Non source code files
cell array of strings

A list of non-source files generated by protected model creation. Examples are `*.mat`, `*.rsp`, and `*.prj`. Default value is empty.

SourceFiles — Source code files
cell array of strings

A list of source code files generated by protected model creation. Examples are `*.c`, `*.h`, `*.cpp`, and `*.hpp`. Default value is empty.

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
string

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

`password` — Password for protected model code generation
string

Password, specified as a string. 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
```

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
string

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

`password` — Password to modify protected model
string

Password, specified as a string. The password is required for modification of the protected model.

See Also
```
Simulink.ModelReference.modifyProtectedModel |
Simulink.ModelReference.protect
```

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
string

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

`password` — Password for protected model simulation
string

Password, specified as a string. 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
```

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
string

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

`password` — Password for read-only view of protected model
string

Password, specified as a string. 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
```

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 **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

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

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

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

More About

- "Define and Display Custom Target Options"
- "Parameter Command-Line Information Summary"
- "Support Optional Features"

See Also
`slConfigUISetEnabled` | `slConfigUISetVal`

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 **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

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

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

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

More About

· "Define and Display Custom Target Options"

· "Parameter Command-Line Information Summary"

· "Support Optional Features"

See Also
`slConfigUIGetVal` | `slConfigUISetVal`

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 **Code Generation** > **Interface** pane of
the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

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

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

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

More About

- "Define and Display Custom Target Options"
- "Parameter Command-Line Information Summary"
- "Support Optional Features"

See Also
slConfigUIGetVal | slConfigUISetEnabled

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);
```

Input Arguments

myConfigObj — Input data
configuration set object

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

Example: `myConfigObj = getActiveConfigSet(model);`

systemTargetFile — Input data
name of system target file

Specify the name of the system target file, such as `ert.tlc` for Embedded Coder, or `grt.tlc` for Generic Real-Time Target coder.

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

Data Types: `char`

targetOptions — Input options
structure of configuration parameter options

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

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

Example: targetOptions.TemplateMakefile = 'grt_default_tmf';

Data Types: struct

More About

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

See Also
getActiveConfigSet | Simulink.ConfigSet | Simulink.ConfigSetRef

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 Simulink Coder build process automatically invokes the Target Language Compiler when generating code. For more information, see "Introduction to the Target Language Compiler".

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

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

You can change the default behavior by specifying one or more compilation *options* as described in "Options" on page 2-156

Options

You can specify one or more compilation options with each `tlc` command. Use spaces to separate options and arguments. TLC resolves options from left to right. If you use conflicting options, the rightmost option prevails. The `tlc` options are:

- "-r Specify Simulink Coder filename" on page 2-157
- "-v Specify verbose level" on page 2-157

- • "-l Specify path to local include files" on page 2-157
- • "-m Specify maximum number of errors" on page 2-157
- • "-O Specify the output file path" on page 2-158
- • "-d[a|c|n|o] Invoke debug mode" on page 2-158
- • "-a Specify parameters" on page 2-158
- • "-p Print progress" on page 2-158
- • "-lint Performance checks and runtime statistics" on page 2-158
- • "-xO Parse only" on page 2-159

-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 Simulink Coder software to set inlining of parameters or file size limits.

-p Print progress

-p *number*

Print a '.' indicating progress for every `number` of TLC primitive operations executed.

-lint Performance checks and runtime statistics

-lint

Perform simple performance checks and collect runtime statistics.

-xO Parse only

`-xO`

Parse only a TLC file; do not execute it.

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

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 Simulink Coder software 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

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
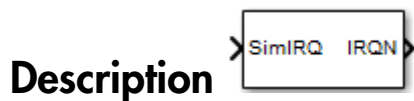
For each specified VxWorks® VME interrupt level, the Async Interrupt block generates an interrupt service routine (ISR) that calls one of the following:

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

You can use the block for simulation and code generation.

Parameters

VME interrupt number(s)

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

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

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

VME interrupt vector offset(s)

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

Simulink task priority(s)

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

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

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

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

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

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

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

Manage own timer

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

Timer resolution (seconds)

The resolution of the ISRs timer. ISRs generated by the Async Interrupt block maintain their own absolute time counters. By default, these timers obtain their values from the VxWorks kernel by using the `tickGet` call. The **Timer resolution** field determines the resolution of these counters. The default resolution is `1/60` second. The `tickGet` resolution for your board support package (BSP) might be different. You should determine the `tickGet` resolution for your BSP and enter it in the **Timer resolution** field.

If you are targeting VxWorks, you can obtain better timer resolution by replacing the `tickGet` call and accessing a hardware timer by using your BSP instead. If you are targeting an RTOS other than VxWorks, you should replace the `tickGet` call with an equivalent call to the target RTOS, or generate code to read the timer register on the target hardware. See " Use Timers in Asynchronous Tasks" and "Async Interrupt Block Implementation" in the Simulink Coder documentation for more information.

Timer size

The number of bits to be used to store the clock tick for a hardware timer. The ISR generated by the Async Interrupt block uses the timer size when you select **Manage own timer**. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the Simulink Coder software determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. However, when **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, the code generator uses a second 32-bit integer to address overflows.

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

Enable simulation input

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

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

Inputs and Outputs

Input

A simulated interrupt source.

Output

Control signal for a

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

Assumptions and Limitations

- The block supports VME interrupts 1 through 7.
- The block requires a VxWorks Board Support Package (BSP) that supports the following VxWorks system calls:
    ```
    sysIntEnable
    sysIntDisable
    intConnect
    intLock
    intUnlock
    tickGet
    ```

Performance Considerations

Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. As

a general rule, it is best to keep ISRs as short as possible. Connect only function-call subsystems that contain a small number of blocks to an Async Interrupt block.

A better solution for large subsystems is to use the Task Sync block to synchronize the execution of the function-call subsystem to a VxWorks task. Place the Task Sync block between the Async Interrupt block and the function-call subsystem. The Async Interrupt block then uses the Task Sync block as the ISR. The ISR releases a synchronization semaphore (performs a `semGive`) to the task, and returns immediately from interrupt level. VxWorks then schedules and runs the task. See the description of the Task Sync block for more information.

See Also

Task Sync
"Handle Asynchronous Events " in the Simulink Coder documentation

Asynchronous Task Specification

Allow for parameter specifications for asynchronous tasks associated with root-level Inport blocks that output a function-call trigger

Library

Asynchronous

Description

The Asynchronous Task Specification block, in combination with a root-level Inport block, allows for an asynchronous function-call input to a model reference.

To implement this feature, place this block at the output port of each root-level Inport block that outputs a function-call trigger. On the **Signal Attributes** pane of the Inport block, select **Output function call** to specify that the Inport block accepts function-call signals. Then use the Asynchronous Task Specification blocks to specify the asynchronous task parameters associated with the respective Inport blocks.

Data Type Support

This specification does not apply to the Asynchronous Task Specification block; the block accepts only function-call signals.

Parameters and Dialog Box

The **Function Block Parameters** dialog box of the Asynchronous Task Specification block appears as follows:
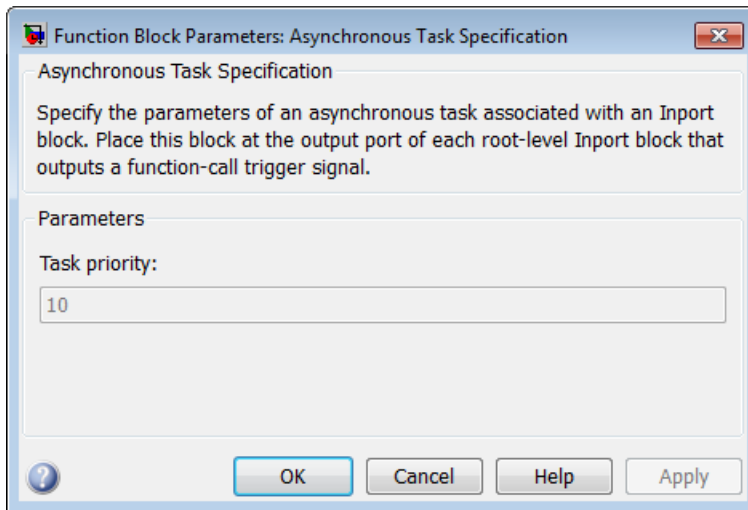
Task priority

Specifies the priority of the asynchronous task calling the destination function-call subsystem. The priority must be a value that generates relevant rate transition behaviors.

Settings

Default: 10

- You can enter an integer or [].
- If you specify an integer for an Asynchronous Task Specification block that resides in a referenced model, the priority of the initiator in the top model mustmatch the priority of that block.
- If you specify [] for an Asynchronous Task Specification block that resides in a referenced model, the priority of the initiator in the top model does not have to match the priority of that block. For this case, the rate transition algorithm is conservative (not optimized), assuming that the priority is unknown but static.

Command-Line Information

This block has only one parameter.

Parameter: `TaskPriority`

Value: `integer`

Configuration Parameters Settings

To create an asynchronous model reference containing a Function-Call and an Asynchronous Task Specification block, you must follow the procedure outlined in "Convert an Asynchronous Subsystem into a Model Reference". One of the steps requires that you make several changes to configuration parameters.

Additional configuration parameters that require attention are the solver **Type** and the **Fixed step size (fundamental sample time)** on the Solver pane. Both the top model and the referenced model must use a fixed-step solver. Moreover, the referenced model must have a fundamental sample time that is an integer multiple of the fundamental sample time of the top model.

Examples

Asynchronous Function-Call Input to Model

This root-level model uses the Inport block with the Asynchronous Task Specification block to allow a function-call input signal to a model reference. The priority is set to 10.
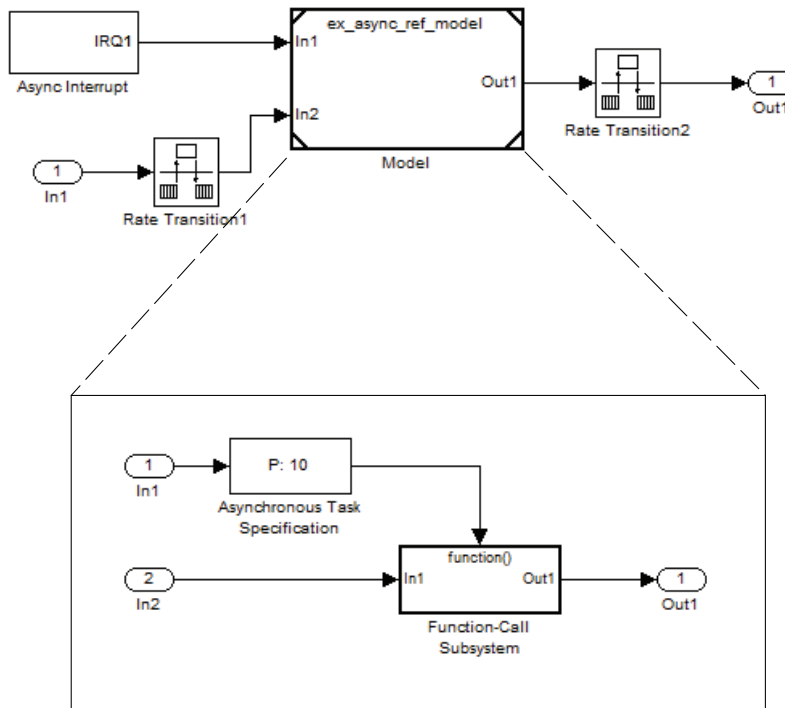
The Asynchronous Task Specification block must immediately follow the Inport block. Also, a branch cannot emanate from the signal connecting the Inport block to the Asynchronous Task Specification block.

Setting Priorities
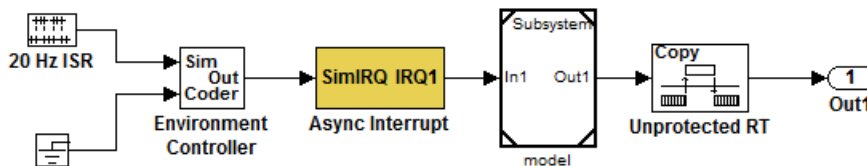
For this model, if the Asynchronous Task Specification block is set to the default value of 10, the Async Interrupt block must also have a priority of 10.
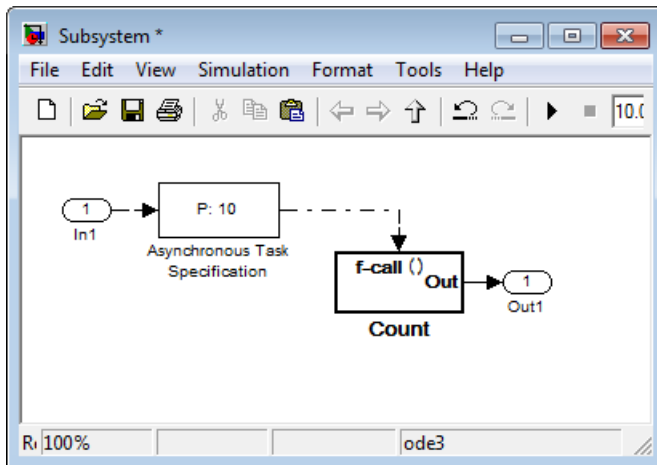
Whereas, if the priority of the Asynchronous Task Specification block is set to the empty matrix, [], the priority of the Async Interrupt can be a value other than 10.
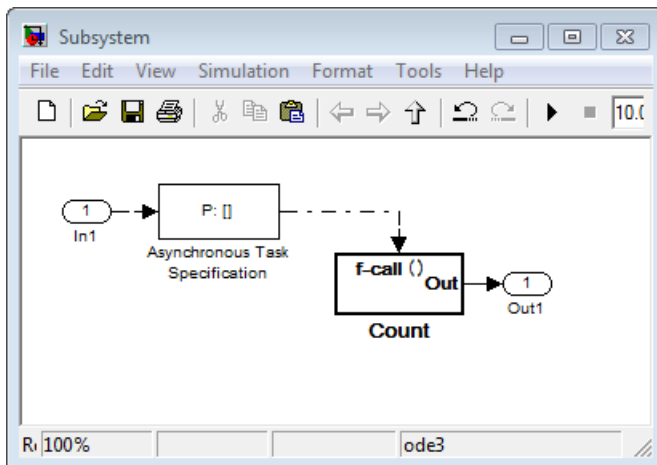
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block

Scalar Expansion	N/A
Dimensionalized	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Function-Call Subsystem block

"Handle Asynchronous Events "

"Model Reference"

Inport block

Generated S-Function

Represent model or subsystem as generated S-function code

Library

S-Function Target

Description
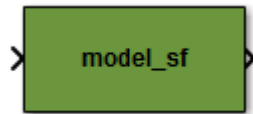
An instance of the Generated S-Function block represents code the Simulink Coder software generates from its S-function target for a model or subsystem. For example, you extract a subsystem from a model and build a Generated S-Function block from it, using the S-function target. This mechanism can be useful for

- Converting models and subsystems to application components
- Reusing models and subsystems
- Optimizing simulation — often, an S-function simulates more efficiently than the original model

For details on how to create a Generated S-Function block from a subsystem, see "Create S-Function Blocks from a Subsystem" in the Simulink Coder documentation.

Requirements

- The S-Function block must perform identically to the model or subsystem from which it was generated.
- Before creating the block, explicitly specify Inport block signal attributes, such as signal widths or sample times. The sole exception to this rule concerns sample times, as described in "Sample Time Propagation in Generated S-Functions".
- You must set the solver parameters of the Generated S-Function block to be the same as those of the original model or subsystem. The generated S-function code

3-13

will operate identically to the original subsystem (see "Choice of Solver Type" in the Simulink Coder documentation for an exception to this rule).

Parameters

Generated S-function name (model_sf)

The name of the generated S-function. The Simulink Coder software derives the name by appending _sf to the name of the model or subsystem from which the block is generated.

Show module list

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

See Also

"Create S-Function Blocks from a Subsystem" in the Simulink Coder documentation

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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

Top of Model Header

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

Bottom of Model Header

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

Example

See "Embed Custom Code Directly Into MdlStart Function".

See Also

Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
"Insert Custom Code Blocks" in the Simulink Coder documentation

Model Source

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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

Top of Model Source

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

Bottom of Model Source

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

Example

See "Embed Custom Code Directly Into MdlStart Function".

See Also

Model Header, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
"Insert Custom Code Blocks" in the Simulink Coder documentation

Protected RT

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

Library

VxWorks (`vxlib1`)

Description
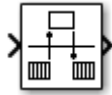
The Protected RT block is a Rate Transition block that is preconfigured to maintain data integrity during data transfers. For more information, see Rate Transition in the Simulink Reference.
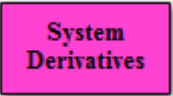
System Derivatives

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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Derivatives Function Declaration Code

Code to be added to the declaration section of the generated `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 Custom Code Blocks" in the Simulink Coder documentation

System Disable

Specify custom system disable code

Library

Custom Code

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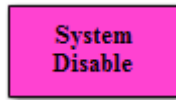
The System Disable block adds user-specified custom code to the declaration, execution, and exit code sections of the SystemDisable function that the code generator creates for the model or subsystem that contains the block.

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

Parameters

System Disable Function Declaration Code

Code to be added to the declaration section of the generated 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 Custom Code Blocks" in the Simulink Coder documentation

System Enable

Specify custom system enable code

Library

Custom Code

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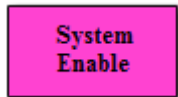
The System Enable block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemEnable` function that the code generator creates for the model or subsystem that contains the block.

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

Parameters

System Enable Function Declaration Code

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

System Enable Function Execution Code

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

System Enable Function Exit Code

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

Example

See "Embed Custom Code Directly Into MdlStart Function".

See Also

Model Header, Model Source, System Derivatives, System Disable, System Initialize, System Outputs, System Start, System Terminate, System Update
"Insert Custom Code Blocks" in the Simulink Coder documentation

System Initialize

Specify custom system initialization code

Library

Custom Code

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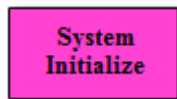
The System Initialize block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemInitialize` function that the code generator creates for the model or subsystem that contains the block.

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

Parameters

System Initialize Function Declaration Code

Code to be added to the declaration section of the generated `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 Custom Code Blocks" in the Simulink Coder documentation

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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Outputs Function Declaration Code

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

System Outputs Function Execution Code

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

System Outputs Function Exit Code

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

Example

See "Embed Custom Code Directly Into MdlStart Function".

See Also

Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Start, System Terminate, System Update
"Insert Custom Code Blocks" in the Simulink Coder documentation

System Start

Specify custom system startup code

Library

Custom Code

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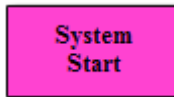
The System Start block adds user-specified custom code to the declaration, execution, and exit code sections of the SystemStart function that the code generator creates for the model or subsystem that contains the block.

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

Parameters

System Start Function Declaration Code

Code to be added to the declaration section of the generated 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 Custom Code Blocks" in the Simulink Coder documentation

System Terminate

Specify custom system termination code

Library

Custom Code

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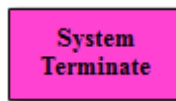
The System Terminate block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemTerminate` function that the code generator creates for the model or subsystem that contains the block.

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

Parameters

System Terminate Function Declaration Code

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

System Terminate Function Execution Code

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

System Terminate Function Exit Code

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

Example

See "Embed Custom Code Directly Into MdlStart Function".

See Also

Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Update
"Insert Custom Code Blocks" in the Simulink Coder documentation

System Update

Specify custom system update code

Library

Custom Code

Description
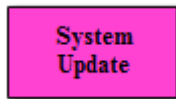
System Update

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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Update Function Declaration Code

Code to be added to the declaration section of the generated `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 Custom Code Blocks" in the Simulink Coder documentation

Task Sync

Spawn VxWorks task to run code of downstream function-call subsystem or Stateflow chart

Library

Asynchronous / Interrupt Templates

Description
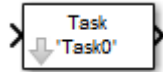
The Task Sync block spawns a VxWorks task that calls a function-call subsystem or Stateflow chart. Typically, you place the Task Sync block between an Async Interrupt block and a function-call subsystem block or Stateflow chart. Alternatively, you might connect the Task Sync block to the output port of a Stateflow diagram that has an event, `Output to Simulink`, configured as a function call.

The Task Sync block performs the following functions:

- Uses the VxWorks system call `taskSpawn` to spawn an independent task. When the task is activated, it calls the downstream function-call subsystem code or Stateflow chart. The block calls `taskDelete` to delete the task during model termination.

- Creates a semaphore to synchronize the connected subsystem with execution of the block.

- Wraps the spawned task in an infinite `for` loop. In the loop, the spawned task listens for the semaphore, using `semTake`. The first call to `semTake` specifies NO_WAIT. This allows the task to determine whether a second `semGive` has occurred prior to the completion of the function-call subsystem or chart. This would indicate that the interrupt rate is too fast or the task priority is too low.

- Generates synchronization code (for example, `semGive()`). This code allows the spawned task to run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. This is accomplished through the connection between the Async Interrupt and Task Sync blocks, which triggers execution of the Task Sync block within an ISR.

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

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

Parameters

Task name (10 characters or less)

The first argument passed to the VxWorks `taskSpawn` system call. VxWorks uses this name as the task function name. This name also serves as a debugging aid; routines use the task name to identify the task from which they are called.

Simulink task priority (0–255)

The VxWorks task priority to be assigned to the function-call subsystem task when spawned. VxWorks priorities range from 0 to 255, with 0 representing the highest priority.

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

Stack size (bytes)

Maximum size to which the task's stack can grow. The stack size is allocated when VxWorks spawns the task. Choose a stack size based on the number of local variables in the task. You should determine the size by examining the generated code for the task (and functions that are called from the generated code).

Synchronize the data transfer of this task with the caller task

If not checked (the default),

- The block maintains a timer that provides absolute time values required by the computations of downstream blocks. The timer is independent of the timer maintained by the Async Interrupt block that calls the Task Sync block.

- A **Timer resolution** option appears.

- The **Timer size** option specifies the word size of the time counter.

If checked,

- The block does not maintain an independent timer, and does not display the **Timer resolution** field.

- Downstream blocks that require timers use the timer maintained by the Async Interrupt block that calls the Task Sync block (see " Use Timers in Asynchronous Tasks" in the Simulink Coder documentation). The timer value is read at the time the asynchronous interrupt is serviced, and data transfers to blocks called by the Task Sync block and execute within the task associated with the Async Interrupt block. Therefore, data transfers are synchronized with the caller.

Timer resolution (seconds)

The resolution of the block's timer in seconds. This option appears only if **Synchronize the data transfer of this task with the caller task** is not checked. By default, the block gets the timer value by calling the VxWorks `tickGet` function. The default resolution is 1/60 second. The `tickGet` resolution for your BSP might be different. You should determine the `tickGet` resolution for your BSP and enter it in the **Timer resolution** field.

Timer size

The number of bits to be used to store the clock tick for a hardware timer. The size can be `32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the Simulink Coder software determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

By default, timer values are stored as 32-bit integers. However, when **Timer size** is `auto`, you can indirectly control the word size of the counters by setting the **Application lifespan (days)** option. If you set **Application lifespan (days)** to a value that is too large for the code generator to handle as a 32-bit integer of the specified resolution, it uses a second 32-bit integer to address overflows.

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

Inputs and Outputs

Input

A call from an Async Interrupt block.

Output

A call to a function-call subsystem.

See Also

Async Interrupt
"Handle Asynchronous Events " in the Simulink Coder documentation

Unprotected RT

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

Library

VxWorks (`vxlib1`)

Description
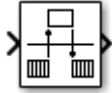
The Unprotected RT block is a Rate Transition block that is preconfigured to conduct deterministic data transfers. For more information, see Rate Transition in the Simulink Reference.

4

Configuration Parameters for Simulink Models

Code Generation Pane: General

The **Code Generation** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

To open the **Code Generation** pane, in the Simulink Editor, select **Simulation > Model Configuration Parameters > Code Generation**.

The **Code Generation** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

In this section...

Code Generation: General Tab Overview

Set up general information about code generation for a model's active configuration set, including target selection, documentation, and build process parameters.

To open the **Code Generation** pane, in the Simulink Editor, select **Simulation > Model Configuration Parameters > Code Generation**.

See Also

"Code Generation Pane: General" on page 4-2

System target file

Specify the system target file.

Settings

Default: `grt.tlc`

You can specify the system target file in these ways:

*   Use the System Target File Browser. Click the **Browse** button, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command.
*   Enter the name of your system target file in this field.

Setting **System target file** to `ert.tlc` displays the **Target hardware** parameter. When you set the **Target hardware** parameter to a specific type of hardware, the Configuration Parameters dialog box displays a **Coder Target** pane for that specific hardware. For more information, see "Target hardware" on page 4-8.

Tips

*   The System Target File Browser lists system target files found on the MATLAB path. Some system target files require additional licensed products.
*   Using ERT-based system target files such as `ert.tlc` to generate code requires an Embedded Coder license.
*   To configure your model for rapid simulation, select `rsim.tlc`.
*   To configure your model for Simulink Real-Time™, select `slrt.tlc` or `slrtert.tlc`.

Command-Line Information
Parameter: `SystemTargetFile`
Type: string
Value: valid system target file
Default: `'grt.tlc'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact ERT based (requires Embedded Coder license)

See Also

"Available Targets"

Browse

Open the System Target File Browser, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command. The value you select is filled into **"System target file"**.

Tips

- The System Target File Browser lists system target files found on the MATLAB path. Some system target files require additional licensed products, such as the Embedded Coder product.
- To configure your model for rapid simulation, select `rsim.tlc`.
- To configure your model for Simulink Real-Time, select `slrt.tlc` or `slrtert.tlc`.

See Also

- "Select a Target"
- "Available Targets"

Language

Specify C or C++ code generation.

Settings

Default: C

C
   Generates C code and places the generated files in your build folder.

C++

Generates C++ code and places the generated files in your build folder.

On the **Code Generation** > **Interface** pane, if you additionally set the **Code interface packaging** parameter to `C++ class`, the build generates a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.

If you set **Code interface packaging** to a value other than `C++ class`, the build generates C++ compatible `.cpp` files containing model interfaces enclosed within an `extern "C"` link directive.

You might need to configure the Simulink Coder software to use a compiler before you build a system.

Dependencies

Selecting `C++` enables and selects the value `C++ class` for the **Code interface packaging** parameter on the **Code Generation** > **Interface** pane.

Command-Line Information
Parameter: `TargetLang`
Type: string
Value: `'C'` | `'C++'`
Default: `'C'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Change Programming Language"

"Compiler or IDE Selection and Configuration"

"Function Prototype Control"

"C++ Class Interface Control"

Description

This field displays the description of the system target file. You can use this description to differentiate between two system target files that have the same file name. To change the value of this description, click the Browse button.

See Also

"Browse" on page 4-6

Target hardware

Select the target hardware for which to generate code.

Note:

- This parameter only appears when the model is configured to use a toolchain-based code generation target, as described in "Configure the Build Process".

- Using this parameter to generate code requires an Embedded Coder license.

To use the **Target hardware** parameter, both of the following actions must be complete:

- Set the **System target file** parameter to `ert.tlc`. This action makes the **Target hardware** parameter visible on the **Code Generation** pane.
- Use Support Package Installer to install the Embedded Coder support package for your target hardware. This action makes target hardware options available for the **Target hardware** parameter.

To install support for your target hardware, set **Target hardware** to `Get more`. This action opens Support Package Installer and displays a list of the support packages that are available for Embedded Coder software.

Settings

Default: None

```
None
```

    Target hardware not specified.

```
Get more...
```

    Select `Get more...` to start Support Package Installer and install Embedded Coder support packages. Embedded Coder support packages add options to the **Target hardware** parameter.

Command-Line Information
Parameter: Not available
Type: Not available
Value: Not available
Default: Not available

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Toolchain"
- "Adding a Custom Toolchain"

Toolchain

Specify the toolchain to use when building an executable or library.

Note: This parameter only appears when the model is configured to use a toolchain-based code generation target, as described in "Configure the Build Process".

Settings

Default: `Automatically locate an installed toolchain`

The list of available toolchains depends on the host computer platform, and can include custom toolchains that you added.

When **Toolchain** is set to `Automatically locate an installed toolchain`, the coder software:

1 Searches your host computer for installed toolchains.
2 Selects the most current toolchain.
3 Displays the name of the selected toolchain immediately below the drop down menu.

Tip

Click **Validate** to verify that the registration information for the toolchain is valid. When the validation process is complete, a separate **Validation report** window opens and displays the results. The Validation report states whether the toolchain registration Passed or Failed and provides status for each step in the validation process. To fix a failure, edit the toolchain definition and repeat the registration process.

Command-Line Information
Parameter: `Toolchain`
Type: string
Value: `'Automatically locate an installed toolchain'` | A valid toolchain name
Default: `'Automatically locate an installed toolchain'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Toolchain"
- "Adding a Custom Toolchain"

Build configuration

Specify compiler optimization or debug settings for toolchain.

Note: This parameter only appears when the model is configured to use a toolchain-based code generation target, as described in "Configure the Build Process".

Settings

Default: `Faster Builds`

`Faster Builds`

> Optimize for shorter build times.

`Faster Runs`

> Optimize for faster-running executable.

`Debug`

> Optimize for debugging.

`Specify`

> Selecting `Specify` displays a table of tools with editable options. Use this table to customize settings for the current model. See "Tool/Options" on page 4-12.

Tip

Click **Show settings** to display a table of tools with options for the current build configuration. See "Tool/Options" on page 4-12.

Customize the toolchain options for the `Specify` build configuration. These options only apply to the current project.

Dependencies

Selecting `Specify` displays a table of tools with editable options. Use this table to customize settings for the current model. See "Tool/Options" on page 4-12.

Command-Line Information
Parameter: `BuildConfiguration`
Type: string
Value: `'Faster Builds'` | `'Faster Runs'` | `'Debug'` | `'Specify'`
Default: `'Faster Builds'`

Recommended Settings

Application	Setting
Debugging	`Debug`
Traceability	No impact
Efficiency	`Faster Runs`
Safety precaution	No impact

See Also

- "Toolchain"
- "Adding a Custom Toolchain"

Tool/Options

Display or customize build configuration settings.

Note: These parameters only appear when the model is configured to use the toolchain approach, as described in "Configure the Build Process"

Settings

The tools column can include: Assembler, C Compiler, Linker, Shared Library Linker, C++ Compiler, C++ Linker, C++ Shared Library Linker, Archiver, Download, Execute, Make Tool. The options can vary by tool and toolchain and can contain macros. Consult third-party toolchain documentation for more information about options you can use with a specific tool.

Dependencies

To display a table of tools and options for the current build configuration, click **Show settings**, next to **Build configuration**.

To create a custom build configuration by editing a table of Tool/Options, set **Build configuration** to `Specify`.

Command-Line Information
Parameter: `CustomToolchainOptions`

Type: string
Value: Specify the baseline toolchain settings. Use a new-line-delineated string to specify each option and its values.
Default: `' '`

See Also

- "Toolchain"
- "Adding a Custom Toolchain"

Compiler optimization level

Control compiler optimizations for building generated code, using flexible, generalized controls.

Note: This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in "Configure the Build Process".

Settings

Default: `Optimizations off (faster builds)`

`Optimizations off (faster builds)`

    Customizes compilation during the Simulink Coder makefile build process to minimize compilation time.

`Optimizations on (faster runs)`

    Customizes compilation during the Simulink Coder makefile build process to minimize run time.

`Custom`

    Allows you to specify custom compiler optimization flags to be applied during the Simulink Coder makefile build process.

Tips

- Target-independent values `Optimizations on (faster runs)` and `Optimizations off (faster builds)` allow you to easily toggle compiler optimizations on and off during code development.

- `Custom` allows you to enter custom compiler optimization flags at Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to Simulink Coder make commands.

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

Dependencies

This parameter enables **Custom compiler optimization flags**.

Command-Line Information
Parameter: `RTWCompilerOptimization`
Type: string
Value: `'Off'` | `'On'` | `'Custom'`
Default: `'Off'`

Recommended Settings

Application	Setting
Debugging	`Optimizations off (faster builds)`
Traceability	`Optimizations off (faster builds)`
Efficiency	`Optimizations on (faster runs)` (execution), No impact (ROM, RAM)
Safety precaution	No impact

See Also

- "Custom compiler optimization flags" on page 4-14
- "Control Compiler Optimizations"

Custom compiler optimization flags

Specify compiler optimization flags to be applied to building the generated code for your model.

Note: This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in "Configure the Build Process".

Settings

Default: `''`

Specify compiler optimization flags without quotes, for example, `-O2`.

Dependency

This parameter is enabled by selecting the value `Custom` for the parameter **Compiler optimization level**.

Command-Line Information
Parameter: `RTWCustomCompilerOptimizations`
Type: string
Value: `''` | user-specified flags
Default: `''`

Recommended Settings

See "Compiler optimization level" on page 4-13.

See Also

- "Compiler optimization level" on page 4-13
- "Control Compiler Optimizations"

Generate makefile

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 the Build Process".

Settings

Default: on

☑ On

   Generates a makefile for a model during the build process.

☐ Off

> Suppresses the generation of a makefile. You must set up post code generation build processing, including compilation and linking, as a user-defined command.

Dependencies

This parameter enables:

- **Make command**
- **Template makefile**

Command-Line Information
Parameter: GenerateMakefile
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Customize Post-Code-Generation Build Processing"
- "Customize Build Process with STF_make_rtw_hook File"
- "Target Development and the Build Process"

Make command

Specify a make command and optionally append 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 the Build Process".

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: string
Value: valid make command MATLAB language file
Default: `'make_rtw'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	`make_rtw`

See Also

- "Template Makefiles and Make Options"
- "Customize Build Process with STF_make_rtw_hook File"
- "Target Development and the Build Process"

Template makefile

Specify the template makefile from which to generate the makefile.

Note: This parameter only appears when the model is configured to use a template makefile-based code generation target, as described in "Configure the Build Process".

Settings

Default: `grt_default_tmf`

The template makefile determines which compiler runs, during the make phase of the build, to compile the generated code. You can specify template makefiles in the following ways:

- Generate a value by selecting a target configuration using the System Target File Browser.
- Explicitly enter a custom template makefile filename (including the extension). The file must be on the MATLAB path.

Tips

- If you do not include a filename extension for a custom template makefile, the code generator attempts to find and execute a MATLAB language file.
- You can customize your build process by modifying an existing template makefile or by providing your own template makefile.

Dependency

This parameter is enabled by **Generate makefile**.

Command-Line Information
Parameter: `TemplateMakefile`
Type: string
Value: valid template makefile filename
Default: `'grt_default_tmf'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

· "Template Makefiles and Make Options"

· "Available Targets"

Ignore custom storage classes

Specify whether to apply or ignore custom storage classes.

Settings

Default: off

☑ On

Ignores custom storage classes by treating data objects that have them as if their storage class attribute is set to `Auto`. Data objects with an `Auto` storage class do not interface with external code and are stored as local or shared variables or in a global data structure.

☐ Off

Applies custom storage classes as specified. You must clear this option if the model defines data objects with custom storage classes.

Tips

- Clear this parameter before configuring data objects with custom storage classes.
- Setting for top-level and referenced models must match.

Dependencies

- This parameter only appears for ERT-based targets.
- Clear this parameter to enable module packaging features.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `IgnoreCustomStorageClasses`
Type: string
Value: `'on'` | `'off`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Custom Storage Classes" in the Embedded Coder documentation

Ignore test point signals

Specify allocation of memory buffers for test points.

Settings

Default: Off

☑ On

> Ignores test points during code generation, allowing optimal buffer allocation for signals with test points, facilitating transition from prototyping to deployment and avoiding accidental degradation of generated code due to workflow artifacts.

☐ Off

> Allocates separate memory buffers for test points, resulting in a loss of code generation optimizations such as reducing memory usage by storing signals in reusable buffers.

Dependencies

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

Command-Line Information

Parameter: `IgnoreTestpoints`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

- "Signals with Test Points" in the Simulink Coder documentation
- "Test Points" in the Simulink documentation
- "Signals" in the Simulink Coder documentation

Select objective

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

Settings

Default: `Unspecified`

`Unspecified`

No objective specified. Do not optimize code generation settings using the Code Generation Advisor.

`Debugging`

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

`Execution efficiency`

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

Tips

For more objectives, specify an ERT-based target.

Dependency

These parameters appear only for GRT-based targets.

Command-Line Information
Parameter: `'ObjectivePriorities'`
Type: cell array of strings
Value: `{''}` | `{'Debugging'}` | `{'Execution efficiency'}`
Default: `{''}`

Recommended Settings

Application	Setting
Debugging	`Debugging`
Traceability	Not applicable for GRT-based targets
Efficiency	`Execution efficiency`

Application	Setting
Safety precaution	Not applicable for GRT-based targets

See Also

- "Application Objectives" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder documentation.

Prioritized objectives

List objectives that you specify by clicking the **Set Objectives** button.

Dependencies

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

Command-Line Information
Command: `get_param('model', 'ObjectivePriorities')`

See Also

- "Application Objectives" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder documentation.

Set Objectives

Open Configuration Set Objectives dialog box.

Dependency

This button appears only for ERT-based targets.

See Also

- "Application Objectives" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder documentation.

Set Objectives — Code Generation Advisor Dialog Box

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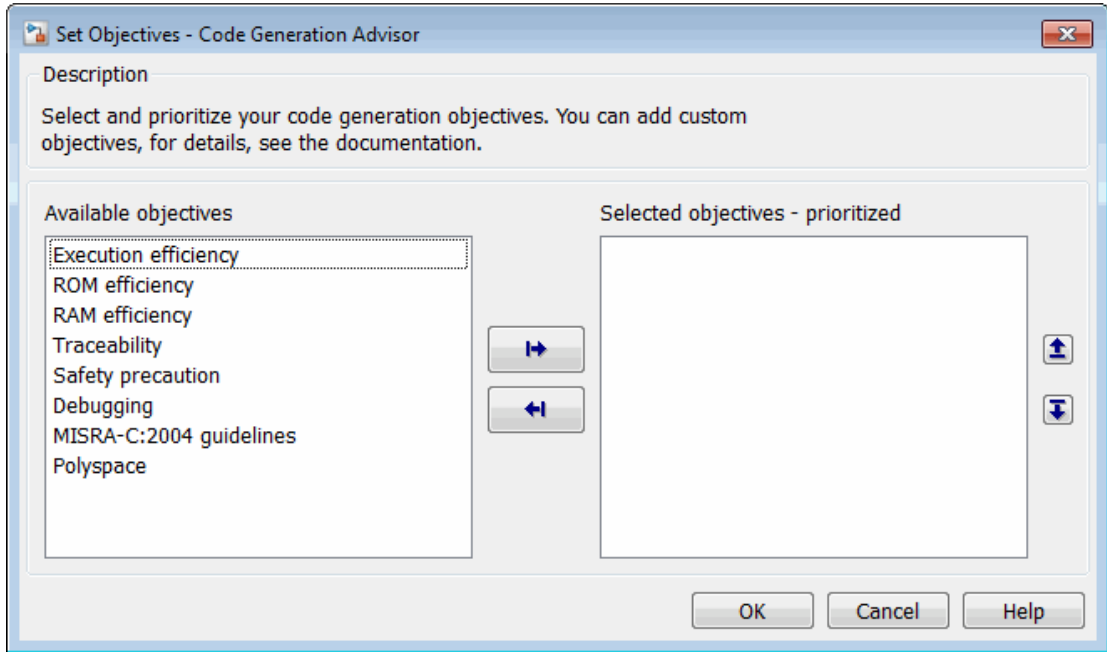
Settings

1 From the **Available objectives** list, select objectives.

2 Click the select button (arrow pointing right) to move the objectives that you selected into the **Selected objectives - prioritized** list.

3 Click the higher priority (up arrow) and lower priority (down arrow) buttons to prioritize the objectives.

Objectives

List of available objectives.
`Execution efficiency` — Configure code generation settings to achieve fast execution time.
`ROM efficiency` — Configure code generation settings to reduce ROM usage.
`RAM efficiency` — Configure code generation settings to reduce RAM usage.
`Traceability` — Configure code generation settings to provide mapping between model elements and code.

`Safety precaution` — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.

`Debugging` — Configure code generation settings to debug the code generation build process.

`MISRA-C:2004 guidelines` — Configure code generation settings to increase compliance with MISRA-C:2004 guidelines.

`Polyspace` — Configure code generation settings to prepare the code for Polyspace® analysis.

Note: If you select the MISRA-C:2004 guidelines code generation objective, the Code Generation Advisor checks:

- The model configuration settings for compliance with the MISRA-C:2004 configuration setting recommendations.

- For blocks that are not supported or recommended for MISRA-C:2004 compliant code generation.

Priorities

After you select objectives from the **Available objectives** parameter, organize the objectives in the **Selected objectives - prioritized** parameter with the highest priority objective at the top.

Dependency

This dialog box appears only for ERT-based targets.

Command-Line Information
Parameter: `'ObjectivePriorities'`
Type: cell array of strings; combination of the available values
Value: `{''}` | `{'Execution efficiency'}` | `{'ROM efficiency'}` | `{'RAM efficiency'}` | `{'Traceability'}` | `{'Safety precaution'}` | `{'Debugging'}` | `{'MISRA-C:2004 guidelines'}` | `{'Polyspace'}`
Default: `{''}`

See Also

- "Application Objectives" in the Embedded Coder documentation.

- "Application Objectives" in the Simulink Coder documentation.

Check Model

Run the Code Generation Advisor checks.

Settings

1 Specify code generation objectives using the **Select objective** parameter (available with GRT-based targets) or in the Configuration Set Objectives dialog box, by clicking **Set Objectives** (available with ERT-based targets).

2 Click **Check Model**. The Code Generation Advisor runs the code generation objectives checks and provide suggestions for changing your model to meet the objectives.

Dependency

You must specify objectives before checking the model.

See Also

- "Application Objectives" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder documentation.

Check model before generating code

Choose whether to run Code Generation Advisor checks before generating code.

Settings

Default: `Off`

`Off`

> Generates code without checking whether the model meets code generation objectives. The code generation report summary and file headers indicate the specified objectives and that the validation was not run.

`On (proceed with warnings)`

> Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the Simulink Coder software continues generating code.

The code generation report summary and file headers indicate the specified objectives and the validation result.

`On (stop for warnings)`

Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the Simulink Coder software does not continue generating code.

Command-Line Information

Parameter: `CheckMdlBeforeBuild`
Type: string
Value: `'Off'` | `'Warning'` | `'Error'`
Default: `'Off'`

Recommended Settings

Application	Setting
Debugging	`On (proceed with warnings)` or `On (stop for warnings)`
Traceability	`On (proceed with warnings)` or `On (stop for warnings)`
Efficiency	`On (proceed with warnings)` or `On (stop for warnings)`
Safety precaution	`On (proceed with warnings)` or `On (stop for warnings)`

See Also

- "Application Objectives" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder documentation.

Generate code only

Specify code generation versus an executable build.

Settings

Default: off

☑ On

> The caption of the **Build/Generate Code** button becomes **Generate Code**. The build process generates code and a makefile, but it does not invoke the make command.

☐ Off

> The caption of the **Build/Generate Code** button becomes **Build**. The build process generates and compiles code, and creates an executable file.

Tip

Generate code only generates a makefile only if you select **Generate makefile**.

Dependency

This parameter changes the function of the **Build/Generate Code** button.

Command-Line Information
Parameter: `GenCodeOnly`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Customize Post-Code-Generation Build Processing"

Build/Generate Code

Start the build or code generation process.

Tip

You can also start the build process by pressing **Ctrl+B**.

Dependency

When you select **Generate code only**, the caption of the **Build** button changes to
Generate Code.

Command-Line Information
Command: rtwbuild
Type: string
Value: '*modelname*'

Recommended Settings

Application	Setting
Debugging	**Build**
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Initiate the Build Process"

Package code and artifacts

Specify whether to automatically package generated code and artifacts for relocation.

Settings

Default: off

☑ On

> The build process runs the packNGo function after code generation to package
> generated code and artifacts for relocation.

☐ Off

The build process does not run `packNGo` after code generation.

Dependency

Selecting this parameter enables **Zip file name** and clearing this parameter disables **Zip file name**.

Command-Line Information
Parameter: `PackageGeneratedCodeAndArtifacts`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

• "Relocate Code to Another Development Environment"
• "packNGo Function Limitations"

Zip file name

Specify the name of the `.zip` file in which to package generated code and artifacts for relocation.

Settings

Default: `' '`

You can enter the name of the `zip` file in which to package generated code and artifacts for relocation. The file name can be specified with or without the `.zip` extension. If you do not specify an extension or an extension other than `.zip`, the `zip` utility adds the `.zip` extension. If a value is not specified, the build process uses the name *model*`.zip`, where *model* is the name of the top model for which code is being generated.

Dependency

This parameter is enabled by **Package code and artifacts**.

Command-Line Information
Parameter: `PackageName`
Type: string
Value: valid name for a `.zip` file
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Relocate Code to Another Development Environment"
- "packNGo Function Limitations"

Code Generation Pane: Report

The **Code Generation** > **Report** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

☐ Create code generation report   ☐ Open report automatically

The **Code Generation** > **Report** pane includes the following additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

Navigation

☐ Code-to-model

☐ Model-to-code   [Configure...]

☐ Generate model Web view

Traceability Report Contents

☐ Eliminated / virtual blocks

☐ Traceable Simulink blocks

☐ Traceable Stateflow objects

☐ Traceable MATLAB functions

Metrics

☐ Static code metrics

☐ Summarize which blocks triggered code replacements

In this section...
"Code Generation: Report Tab Overview" on page 4-33
"Create code generation report" on page 4-33
"Open report automatically" on page 4-36
"Code-to-model" on page 4-37
"Model-to-code" on page 4-38

In this section...
"Configure" on page 4-40
"Generate model Web view" on page 4-40
"Eliminated / virtual blocks" on page 4-41
"Traceable Simulink blocks" on page 4-42
"Traceable Stateflow objects" on page 4-43
"Traceable MATLAB functions" on page 4-44
"Static code metrics" on page 4-45
"Summarize which blocks triggered code replacements" on page 4-46

Code Generation: Report Tab Overview

Control the code generation report that the Simulink Coder software automatically creates.

Configuration

To create a code generation report during the build process, select the **Create code generation report** parameter.

See Also

- "Generate a Code Generation Report"
- "Reports for Code Generation"

  If you have an Embedded Coder license, see also "HTML Code Generation Report Extensions".
- "Code Generation Pane: Report" on page 4-32

Create code generation report

Document generated code in an HTML report.

Settings

Default: Off

☑ On

Generates a summary of code generation source files in an HTML report. Places the report files in an `html` subfolder within the build folder. In the report,

- The **Summary** section lists version and date information. The **Configuration Settings at the Time of Code Generation** link opens a noneditable view of the Configuration Parameters dialog that shows the Simulink model settings, including TLC options, at the time of code generation.

- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.

- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data (requires an Embedded Coder license and the ERT target).

- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**, providing a complete mapping between model elements and code (requires an Embedded Coder license and the ERT target).

- The **Static Code Metrics Report** section provides statistics of the generated code. Metrics are estimated from static analysis of the generated code.

- The **Code Replacements Report** section allows you to account for code replacement library (CRL) functions that were used during code generation, providing a mapping between each replacement instance and the Simulink block that triggered the replacement.

In the **Generated Files** section, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code,

- Global variable instances are hyperlinked to their definitions.

- If you selected the traceability option **Code-to-model**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view the relevant blocks or subsystems in a Simulink model window (requires an Embedded Coder license and the ERT target).

- If you selected the traceability option **Model-to-code**, you can view the generated code for a block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **C/C++ Code** > **Navigate to C/C++ Code** (requires an Embedded Coder license and the ERT target).

- If you set the **Code coverage tool** parameter on the **Code Generation** > **Verification** pane, you can view the code coverage data and annotations in the generated code in the HTML Code Generation Report (requires an Embedded Coder license and the ERT target).

☐ Off

Does not generate a summary of files.

Dependency

This parameter enables and selects

- "Open report automatically" on page 4-36
- "Code-to-model" on page 4-37 (ERT target)

This parameter enables

- "Model-to-code" on page 4-38 (ERT target)
- "Eliminated / virtual blocks" on page 4-41 (ERT target)
- "Traceable Simulink blocks" on page 4-42 (ERT target)
- "Traceable Stateflow objects" on page 4-43 (ERT target)
- "Traceable MATLAB functions" on page 4-44 (ERT target)

.

Command-Line Information
Parameter: `GenerateReport`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"Reports for Code Generation"

If you have an Embedded Coder license, see also "HTML Code Generation Report Extensions".

If you have an Embedded Coder license, see also "Code Coverage in SIL and PIL Simulations".

Open report automatically

Specify whether to display code generation reports automatically.

Settings
Default: Off

☑ On

>   Displays the code generation report automatically in a new browser window.

☐ Off

>   Does not display the code generation report, but the report is still available in the `html` folder.

Dependency

This parameter is enabled and selected by **Create code generation report**.

Command-Line Information
Parameter: LaunchReport
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

"Reports for Code Generation"

If you have an Embedded Coder license, see also "HTML Code Generation Report Extensions".

Code-to-model

Include hyperlinks in the code generation report that link code to the corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram.

Settings

Default: Off

☑ On

> Includes hyperlinks in the code generation report that link code to corresponding Simulink blocks, Stateflow objects, and MATLAB functions in the model diagram. The hyperlinks provide traceability for validating generated code against the source model.

☐ Off

> Omits hyperlinks from the generated report.

Tip

Clear this parameter to speed up code generation. For large models (containing over 1000 blocks), generation of hyperlinks can be time consuming.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled and selected by **Create code generation report**.

• You must select **Include comments** on the **Code Generation** > **Comments** pane to use this parameter.

Command-Line Information

Parameter: `IncludeHyperlinkInReport`
Type: string
Value: `'on'` | `'off`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"HTML Code Generation Report Extensions"

Model-to-code

Link Simulink blocks, Stateflow objects, and MATLAB functions in a model diagram to corresponding code segments in a generated HTML report so that the generated code for a block can be highlighted on request.

Settings

Default: Off

☑ On

Includes model-to-code highlighting support in the code generation report. To highlight the generated code for a Simulink block, Stateflow object, or MATLAB script in the code generation report, right-click the item and select **C/C++ Code** > **Navigate to C/C++ Code**.

☐ Off

Omits model-to-code highlighting support from the generated report.

Tip

Clear this parameter to speed up code generation. For large models (containing over 1000 blocks), generation of model-to-code highlighting support can be time consuming.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.
- This parameter selects:
  - "Eliminated / virtual blocks" on page 4-41
  - "Traceable Simulink blocks" on page 4-42
  - "Traceable Stateflow objects" on page 4-43
  - "Traceable MATLAB functions" on page 4-44
- You must select the following parameters to use this parameter:
  - "Include comments" on the **Code Generation** > **Comments** pane
  - At least one of the following:
    - "Eliminated / virtual blocks" on page 4-41
    - "Traceable Simulink blocks" on page 4-42
    - "Traceable Stateflow objects" on page 4-43
    - "Traceable MATLAB functions" on page 4-44

Command-Line Information

Parameter: `GenerateTraceInfo`
Type: Boolean
Value: `on` | `off`
Default: `off`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	On

See Also

"HTML Code Generation Report Extensions"

Configure

Open the **Model-to-code navigation** dialog box. This dialog box provides a way for you to specify a build folder containing previously-generated model code to highlight. Applying your build folder selection will attempt to load traceability information from the earlier build, for which **Model-to-code** must have been selected.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by "Model-to-code" on page 4-38.

See Also

"HTML Code Generation Report Extensions"

Generate model Web view

Include the model Web view in the code generation report to navigate between the code and model within the same window. You can share your model and generated code outside of the MATLAB environment. You must have a Simulink Report Generator™ license to include a "Web view" of the model in the code generation report.

Settings

Default: Off

☑ On

　Include model Web view in the code generation report.

☐ Off

Omit model Web view in the code generation report.

Command-Line Information
Parameter: `GenerateWebview`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Web View of Model in Code Generation Report"

Eliminated / virtual blocks

Include summary of eliminated and virtual blocks in code generation report.

Settings

Default: Off

☑ On

 Includes a summary of eliminated and virtual blocks in the code generation report.

☐ Off

 Does not include a summary of eliminated and virtual blocks.

Dependencies

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

- This parameter is selected by **Model-to-code**.

Command-Line Information
Parameter: `GenerateTraceReport`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"HTML Code Generation Report Extensions"

Traceable Simulink blocks

Include summary of Simulink blocks in code generation report.

Settings

Default: Off

☑ On

Includes a summary of Simulink blocks and the corresponding code location in the code generation report.

☐ Off

Does not include a summary of Simulink blocks.

Dependencies

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

- This parameter is enabled by **Create code generation report**.
- This parameter is selected by **Model-to-code**.

Command-Line Information
Parameter: `GenerateTraceReportSl`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"HTML Code Generation Report Extensions"

Traceable Stateflow objects

Include summary of Stateflow objects in code generation report.

Settings

Default: Off

☑ On

Includes a summary of Stateflow objects and the corresponding code location in the code generation report.

☐ Off

Does not include a summary of Stateflow objects.

Dependencies

- This parameter only appears for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.
- This parameter is selected by **Model-to-code**.

Command-Line Information
Parameter: `GenerateTraceReportSf`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"HTML Code Generation Report Extensions"

"Traceability of Stateflow Objects in Generated Code"

Traceable MATLAB functions

Include summary of MATLAB functions in code generation report.

Settings

Default: Off

☑ On

> Includes a summary of MATLAB functions and corresponding code locations in the code generation report.

☐ Off

> Does not include a summary of MATLAB functions.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Create code generation report**.
- This parameter is selected by **Model-to-code**.

Command-Line Information
Parameter: `GenerateTraceReportEml`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"HTML Code Generation Report Extensions"

Static code metrics

Include static code metrics report in the code generation report.

Settings

Default: Off

☑ On

　　Include static code metrics report in the code generation report. The static code
　　metrics report does not support the C++ target language.

☐ Off

　　Omit static code metrics report from the code generation report.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.

Command-Line Information
Parameter: GenerateCodeMetricsReport
Type: Boolean
Value: on | off
Default: off

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Static Code Metrics"

Summarize which blocks triggered code replacements

Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

Settings

Default: Off

☑ On

Include code replacement report in the code generation report.

Note: Selecting this option also generates code replacement trace information for viewing in the **Trace Information** tab of the Code Replacement Viewer. The

generated information can help you determine why an expected code replacement did not occur.

☐ Off

Omit code replacement report from the code generation report.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.

Command-Line Information
Parameter: `GenerateCodeReplacementReport`
Type: Boolean
Value: `on` | `off`
Default: `off`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Analyze Code Replacements in the Generated Code"
- "Trace Code Replacements Generated Using Your Code Replacement Library"
- "Determine Why Code Replacement Functions Were Not Used"

Code Generation Pane: Comments

The **Code Generation** > **Comments** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

```
┌─ Overall control ──────────────────────────────────────────────────┐
│ ☑ Include comments                                                  │
└─────────────────────────────────────────────────────────────────────┘
┌─ Auto generated comments ──────────────────────────────────────────┐
│ ☑ Simulink block / Stateflow object comments                        │
│ ☐ MATLAB source code as comments                                    │
│ ☐ Show eliminated blocks                                            │
│ ☐ Verbose comments for SimulinkGlobal storage class                 │
└─────────────────────────────────────────────────────────────────────┘
```

The **Code Generation** > **Comments** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

```
┌─ Overall control ──────────────────────────────────────────────────┐
│ ☑ Include comments                                                  │
└─────────────────────────────────────────────────────────────────────┘
┌─ Auto generated comments ──────────────────────────────────────────┐
│ ☑ Simulink block / Stateflow object comments                        │
│ ☐ MATLAB source code as comments                                    │
│ ☐ Show eliminated blocks                                            │
│ ☐ Verbose comments for SimulinkGlobal storage class                 │
│ ☐ Operator annotations                                              │
└─────────────────────────────────────────────────────────────────────┘
┌─ Custom comments ──────────────────────────────────────────────────┐
│ ☐ Simulink block descriptions        ☐ Stateflow object descriptions │
│ ☐ Simulink data object descriptions  ☐ Requirements in block comments │
│ ☐ Custom comments (MPT objects only) ☐ MATLAB function help text      │
└─────────────────────────────────────────────────────────────────────┘
```

Code Generation: Comments Tab Overview

Control the comments that the Simulink Coder software automatically creates and inserts into the generated code.

See Also

"Code Generation Pane: Comments" on page 4-48

Include comments

Specify which comments are in generated files.

Settings

Default: on

☑ On

Places comments in the generated files based on the selections in the **Auto generated comments** pane.

☐ Off

Omits comments from the generated files.

Note: This parameter does not apply to copyright notice comments in the generated code.

Dependencies

This parameter enables:

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

Command-Line Information
Parameter: `GenerateComments`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

Simulink block / Stateflow object comments

Specify whether to insert Simulink block and Stateflow object comments.

Settings

Default: on

☑ On

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

☐ Off

> Suppresses comments.

Dependency

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

Command-Line Information
Parameter: `SimulinkBlockComments`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

MATLAB source code as comments

Specify whether to insert MATLAB source code as comments.

Settings

Default: off

☑ On

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

4-51

Includes the function signature in the function banner.

☐ Off

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

Dependency

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

Command-Line Information
Parameter: `MATLABSourceComments`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"Include MATLAB Code as Comments in Generated Code"

Show eliminated blocks

Specify whether to insert eliminated block's comments.

Settings

Default: off

☑ On

Inserts statements in the generated code from blocks eliminated as the result of optimizations (such as parameter inlining).

☐ Off

    Suppresses statements.

Dependency

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

Command-Line Information
Parameter: ShowEliminatedStatement
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

Verbose comments for SimulinkGlobal storage class

You can control the generation of comments in the model parameter structure declaration in *model*_prm.h. Parameter comments indicate parameter variable names and the names of source blocks.

Settings

Default: off

☑ On

    Generates parameter comments regardless of the number of parameters.

☐ Off

    Generates parameter comments if less than 1000 parameters are declared. This reduces the size of the generated file for models with a large number of parameters.

Dependency

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

Command-Line Information
Parameter: ForceParamTrailComments
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

Operator annotations

Specify whether to include operator annotations for Polyspace in the generated code as comments.

Settings

Default: Off

☑ On

Includes operator annotations in the generated code.

☐ Off

Does not include operator annotations in the generated code.

Tips

- These annotations help document overflow behavior that is due to the way the Embedded Coder software implements an operation. These operators cannot be traced to an overflow in the design.

- Justify operators that the Polyspace software cannot prove. When this option is enabled, if the Embedded Coder software uses one of these operators, it adds annotations to the generated code to justify the operators for Polyspace.
- Embedded Coder cannot justify operators that result from the design.

Dependency

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

Command-Line Information
Parameter: `OperatorAnnotations`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"Code Annotation for Justifying Polyspace Checks"

Simulink block descriptions

Specify whether to insert descriptions of blocks into generated code as comments.

Settings

Default: off

☑ On

Includes the following comments in the generated code for each block in the model, with the exception of virtual blocks and blocks removed due to block reduction:

- The block name at the start of the code, regardless of whether you select **Simulink block / Stateflow object comments**
- Text specified in the **Description** field of each Block Properties dialog box

The block names and descriptions can include international (non-US-ASCII) characters.

☐ Off

Suppresses the generation of block name and description comments in the generated code.

Dependency

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

Command-Line Information

Parameter: InsertBlockDesc
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

"International Character Support"

Simulink data object descriptions

Specify whether to insert descriptions of Simulink data objects into generated code as comments.

Settings

Default: off

☑ On

> Inserts contents of the **Description** field in the Model Explorer Object Properties pane for each Simulink data object (signal, parameter, and bus objects) in the generated code as comments.
>
> The descriptions can include international (non-US-ASCII) characters.

☐ Off

> Suppresses the generation of data object property descriptions as comments in the generated code.

Dependency

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

Command-Line Information
Parameter: `SimulinkDataObjDesc`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

Custom comments (MPT objects only)

Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.

Settings

Default: off

☑ On

> Inserts comments just above the identifiers for signal and parameter MPT objects in generated code.

☐ Off

> Suppresses the generation of custom comments for signal and parameter identifiers.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter requires that you include the comments in a function defined in a MATLAB language file or TLC file that you specify with **Custom comments function**.

Command-Line Information
Parameter: `EnableCustomComments`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

"Add Custom Comments for Signal or Parameter Identifiers"

Custom comments function

Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects

Settings

Default: ' '

Enter the name of the MATLAB language file or TLC file for the function that includes the comments to be inserted of your MPT signal and parameter objects. You can specify the file name directly or click **Browse** and search for a file.

Tip

You might use this option to insert comments that document some or all of the property values of an object.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Custom comments (MPT objects only)**.

Command-Line Information
Parameter: CustomCommentsFcn
Type: string
Value: valid file name
Default: ' '

Recommended Settings

Application	Setting
Debugging	Valid file name
Traceability	Valid file name
Efficiency	No impact
Safety precaution	No impact

See Also

"Add Custom Comments for Signal or Parameter Identifiers"

Stateflow object descriptions

Specify whether to insert descriptions of Stateflow objects into generated code as comments.

Settings

Default: off

☑ On

> Inserts descriptions of Stateflow states, charts, transitions, and graphical functions into generated code as comments. The descriptions come from the **Description** field in Object Properties pane in the Model Explorer for these Stateflow objects. The comments appear just above the code generated for each object.
>
> The descriptions can include international (non-US-ASCII) characters.

☐ Off

> Suppresses the generation of comments for Stateflow objects.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Stateflow license.

Command-Line Information
Parameter: `SFDataObjDesc`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

"International Character Support"

Requirements in block comments

Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.

Settings

Default: off

☑ On

Inserts the requirement descriptions that you assign to Simulink blocks into the generated code as comments. The Simulink Coder software includes the requirement descriptions in the generated code in the following locations.

Model Element	Requirement Description Location
Model	In the main header file *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

The requirement text can include international (non-US-ASCII) characters.

☐ Off

Suppresses the generation of comments for block requirement descriptions.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires Embedded Coder and Simulink Verification and Validation™ licenses when generating code.

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: string
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	On

See Also

"How Requirements Information Is Included in Generated Code" in the Simulink Verification and Validation documentation

MATLAB function help text

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

Settings

Default: off

☑ On

  Inserts MATLAB function help text in the function banner.

☐ Off

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

Dependency

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

Command-Line Information
Parameter: `MATLABFcnDesc`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

"Including MATLAB Function Help Text in the Function Banner"

Code Generation Pane: Symbols

The **Code Generation** > **Symbols** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

Auto-generated identifier naming rules

Maximum identifier length: 31

Reserved names

☐ Use the same reserved names as Simulation Target

Reserved names:

The **Code Generation** > **Symbols** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

Auto-generated identifier naming rules

Identifier format control

Global variables:	RN$M
Global types:	NR$M_T
Field name of global types:	NM
Subsystem methods:	RNMF
Subsystem method arguments:	rt$I$N$M
Local temporary variables:	NM
Local block output variables:	rtb_NM
Constant macros:	RN$M
Shared utilities:	NC

Minimum mangle length:	1
Maximum identifier length:	31
System-generated identifiers:	Shortened ▼
Generate scalar inlined parameters as:	Literals ▼

Simulink data object naming rules

Signal naming:	None ▼
Parameter naming:	None ▼
#define naming:	None ▼

Reserved names

☐ Use the same reserved names as Simulation Target

Reserved names:

Code Generation: Symbols Tab Overview

Select the automatically generated identifier naming rules.

See Also

- "Code Generation Pane: Symbols" on page 4-64
- "Construction of Generated Identifiers"
- "Identifier Name Collisions and Mangling"
- "Specify Identifier Length to Avoid Naming Collisions"

- "Specify Reserved Names for Generated Identifiers"
- "Customize Generated Identifier Naming Rules" in the Embedded Coder documentation

Global variables

Customize generated global variable identifiers.

Settings

Default: RN$M

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$M	Insert name mangling string if required to avoid naming collisions. Required.
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify $R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the $R and $M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator

preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

- This parameter setting only determines the name of objects, such as signals and parameters, if the object is set to Auto.

Dependency

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

Command-Line Information
Parameter: CustomSymbolStrGlobalVar
Type: string
Value: valid combination of tokens
Default: RN$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	RN$M

See Also

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Avoid Identifier Name Collisions with Referenced Models" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Global types

Customize generated global type identifiers.

Settings

Default: NR$M_T

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$M	Insert name mangling string if required to avoid naming collisions. Required.
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify $R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the $R and $M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, typedef statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify $R, the code generator includes the model name in the typedef.

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

Dependency

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

Command-Line Information
Parameter: `CustomSymbolStrType`
Type: string
Value: valid combination of tokens
Default: `$N$R$M_T`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	`$N$R$M_T`

See Also

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Avoid Identifier Name Collisions with Referenced Models" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Field name of global types

Customize generated field names of global types.

Settings

Default: `$N$M`

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$A	Insert data type acronym into signal and work vector identifiers. For example, i32 for int32_t.
$H	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string 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 string if required to avoid naming collisions. Required.
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- The **Maximum identifier length** setting does not apply to type definitions.
- This option does not impact objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Dependency

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

Command-Line Information
Parameter: CustomSymbolStrField
Type: string
Value: valid combination of tokens
Default: NM

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	NM

See Also

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Subsystem methods

Customize generated function names for reusable subsystems.

Settings

Default: RNMF

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$F	Insert method name (for example, _Update for update method).
$H	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string 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 string if required to avoid naming collisions. Required.

Token	Description
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character.
	Required for model referencing.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.

- If you specify `$R`, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens.

- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.

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

Dependency

- This parameter appears only for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `CustomSymbolStrFcn`
Type: string
Value: valid combination of tokens
Default: RNMF

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	RNMF

See Also

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Avoid Identifier Name Collisions with Referenced Models" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Subsystem method arguments

Customize generated function argument names for reusable subsystems.

Settings

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated argument name. The macro string can include a combination of the following format tokens.

Token	Description
$I	Insert an u if the argument is an input. Insert a y if the argument is an output. Optional.
$M	Insert name mangling string if required to avoid naming collisions. Required.

Token	Description
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. Recommended to maximize readability of generated code.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.

Dependencies

This parameter:

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

Command-Line Information
Parameter: `CustomSymbolStrFcnArg`
Type: string
Value: valid combination of tokens
Default: `rt$I$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combinations of tokens
Efficiency	No impact
Safety precaution	`rt$I$N$M`

See Also

- "Code Generation Pane: Symbols" on page 4-64

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Local temporary variables

Customize generated local temporary variable identifiers.

Settings

Default: NM

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$A	Insert data type acronym (for example, i32 for integers) into signal and work vector identifiers.
$M	Insert name mangling string if required to avoid naming collisions. Required.
$N	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated.
$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- Avoid name collisions. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate. Reserve at least three characters for a name mangling string.

- If you specify $R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the $R and $M tokens.

- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

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

Dependency

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

Command-Line Information

Parameter: CustomSymbolStrTmpVar
Type: string
Value: valid combination of tokens
Default: NM

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	NM

See Also

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Avoid Identifier Name Collisions with Referenced Models" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Local block output variables

Customize generated local block output variable identifiers.

Settings

Default: `rtb_$N$M`

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$A	Insert data type acronym (for example, `i32` for integers) into signal and work vector identifiers.
$M	Insert name mangling string if required to avoid naming collisions. Required.
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- This option does not impact objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

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

Command-Line Information
Parameter: `CustomSymbolStrBlkIO`
Type: string

Value: valid combination of tokens
Default: `rtb_$N$M`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	`rtb_$N$M`

See Also

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Constant macros

Customize generated constant macro identifiers.

Settings

Default: `$R$N$M`

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$M	Insert name mangling string if required to avoid naming collisions. Required.
$N	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.

Token	Description
$R	Insert root model name into identifier, replacing unsupported characters with the underscore (_) character. Required for model referencing.

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify $R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the $R and $M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- This option does not impact objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Dependency

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

Command-Line Information
Parameter: CustomSymbolStrMacro
Type: string
Value: valid combination of tokens
Default: RN$M

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens

Application	Setting
Efficiency	No impact
Safety precaution	RN$M

See Also

- "Identifier Format Control" in the Embedded Coder documentation
- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Avoid Identifier Name Collisions with Referenced Models" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Shared utilities

Customize shared utility identifiers.

Settings

Default: NC

Customize generated shared utility identifier names.

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
$N	Insert name of object (block, signal or signal object, state, parameter, or parameter object) for which identifier is generated. Optional.
$C	Insert eight-character conditional checksum when $N is not specified or the **Maximum identifier length** does not accommodate the full length of $N. Required.

Tips

- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers that you expect to generate.

- The checksum token $C is required. If $C is specified without $N, the checksum is included in the identifier name. Otherwise, the code generator includes the checksum when necessary to prevent name collisions.

- If you specify $N, then the checksum is only included in the name when the identifier length is too short to accommodate the fully expanded format string. The code generator includes the checksum and truncates $N until the length is equal to **Maximum identifier length**. When necessary, an underscore is inserted to separate tokens.

- Descriptive text helps make the identifier name more accessible.

Dependency

- This parameter appears only for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: CustomSymbolStrUtil
Type: string
Value: valid combination of tokens
Default: NC

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid combination of tokens
Efficiency	No impact
Safety precaution	NC

See Also

- "Code Generation Pane: Symbols"
- "Identifier Format Control"
- "Exceptions to Identifier Formatting Conventions"

Minimum mangle length

Increase the minimum number of characters for generating name mangling strings to help avoid name collisions.

Settings

Default: 1

Specify an integer value that indicates the minimum number of characters the code generator uses when generating a name mangling string. The maximum possible value is 15. The minimum value automatically increases during code generation as a function of the number of collisions. A larger value reduces the chance of identifier disturbance when you modify the model.

Tips

- Minimize disturbance to the generated code during development by specifying a value of 4. This value is conservative. It allows for over 1.5 million collisions for a particular identifier before the mangle length increases.

- Set the value to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

Dependency

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

Command-Line Information
Parameter: MangleLength
Type: integer
Value: value between 1 and 15
Default: 1

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	1
Efficiency	No impact
Safety precaution	No impact

See Also

- "Control Name Mangling in Generated Identifiers" in the Embedded Coder documentation
- "Maintain Traceability for Generated Identifiers" in the Embedded Coder documentation

Maximum identifier length

Specify maximum number of characters in generated function, type definition, variable names.

Settings

Default: 31
Minimum: 31
Maximum: 256

You can use this parameter to limit the number of characters in function, type definition, and variable names.

Tips

- Consider increasing identifier length for models having a deep hierarchical structure.
- When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate the root model name, and possibly, the name mangling string. A code generation error occurs if **Maximum identifier length** is too small.
- This parameter must be the same for both top-level and referenced models.
- When a name conflict occurs between a symbol within the scope of a higher level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. Name mangling is performed on the symbol from the higher level model.

Command-Line Information
Parameter: MaxIdLength
Type: integer
Value: valid value
Default: 31

Recommended Settings

Application	Setting
Debugging	Valid value
Traceability	>30
Efficiency	No impact
Safety precaution	>30

See Also

- "Construction of Generated Identifiers"
- "Identifier Name Collisions and Mangling"
- "Identifier Format Control" in the Embedded Coder documentation

System-generated identifiers

Specify whether the code generator uses shorter, more consistent names for the $N token in system-generated identifiers.

Settings

Default: Shortened

Classic

Generate longer identifier names, which are used by default before R2013a, for the $N token. For example, for a model named sym, if:

- "Global variables" is NR$M, the block state identifier is sym_DWork.
- "Global types" is RN$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" is NR$M, the block state identifier is sym_DW.
- "Global types" is RN$M, the block state type is a structure named DW_sym.

System-generated identifiers per model

Classic	Shortened	Data Representation	Description
`BlockIO, B`	`B`	Type	Block signals of the system
`ExternalInputs`	`ExtU`	Type	Block input data for root system
`ExternalInputSizes`	`ExtUSize`	Type	Size of block input data for the root system (used when inputs are variable dimensions)
`ExternalOutputs`	`ExtY`	Type	Block output data for the root system
`ExternalOutputSizes`	`ExtYSize`	Type	Size of block output data for the root system
`Parameters`	`P`	Type	Parameters for the system
`ConstBlockIO`	`ConstB`	`Const` Type	Block inputs and outputs that are constants
`MachineLocalData, Machine`	`MachLocal`	`Const` Type, Global Variable	Used by ERT S-function targets
`ConstParam, ConstP`	`ConstP`	`Const` Type, Global Variable	Constant parameters in the system
`ConstParamWithInit, ConstWithInitP`	`ConstInitP`	`Const` Type, Global Variable	Initialization data for constant parameters in the system
`D_Work, DWork`	`DW`	Type, Global Variable	Block states in the system
`MassMatrixGlobal`	`MassMatrix`	Type, Global Variable	Used for physical modeling blocks
`PrevZCSigStates, PrevZCSigState`	`PrevZCX`	Type, Global Variable	Previous zero-crossing signal state
`ContinuousStates, X`	`X`	Type, Global Variable	Continuous states

Classic	Shortened	Data Representation	Description
`StateDisabled, Xdis`	`XDis`	Type, Global Variable	Status of an enabled subsystem
`StateDerivatives, Xdot`	`XDot`	Type, Global Variable	Derivatives of continuous states at each time step
`ZCSignalValues, ZCSignalValue`	`ZCV`	Type, Global Variable	Zero-crossing signals
`DefaultParameters`	`DefaultP`	Global Variable	Default parameters in the system
`GlobalTID`	`GlobalTID`	Global Variable	Used for sample time for states in referenced models
`InvariantSignals`	`Invariant`	Global Variable	Invariant signals
`NSTAGES`	`NSTAGES`	Global Variable	Solver macro
`Object`	`Obj`	Global Variable	Used by ERT C++ code generation to refer to referenced model objects
`TimingBridge`	`TimingBrdg`	Global Variable	Timing information stored in different data structures
`U`	`U`	Global Variable	Input data
`USize`	`USize`	Global Variable	Size of input data
`Y`	`Y`	Global Variable	Output data
`YSize`	`YSize`	Global Variable	Size of output data

System-generated identifier names per referenced model or reusable subsystem

Classic	Shortened	Data Representation	Description
`rtB, B`	`B`	Type, Global Variable	Block signals of the system

Classic	Shortened	Data Representation	Description
`rtC, C`	`ConstB`	Type, Global Variable	Block inputs and outputs that are constants
`rtDW, DW`	`DW`	Type, Global Variable	Block states in the system
`rtMdlrefDWork, MdlrefDWork`	`MdlRefDW`	Type, Global Variable	Block states in referenced model
`rtP, P`	`P`	Type, Global Variable	Parameters for the system
`rtRTM, RTM`	`RTM`	Type, Global Variable	RT_Model structure
`rtX, X`	`X`	Type, Global Variable	Continuous states in model reference
`rtXdis, Xdis`	`XDis`	Type, Global Variable	Status of an enabled subsystem
`rtXdot, Xdot`	`XDot`	Type, Global Variable	Derivatives of the S-function's continuous states at each time step
`rtZCE, ZCE`	`ZCE`	Type, Global Variable	Zero-crossing enabled
`rtZCSV, ZCSV`	`ZCV`	Type, Global Variable	Zero-crossing signal values

Dependencies

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

Command-Line Information
Parameter: `InternalIdentifier`
Type: string
Value: `Classic` | `Shortened`
Default: `Shortened`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Construction of Generated Identifiers"
- "Identifier Name Collisions and Mangling"
- "Specify Identifier Length to Avoid Naming Collisions"
- "Specify Reserved Names for Generated Identifiers"
- "Customize Generated Identifier Naming Rules" in the Embedded Coder documentation
- "Identifier Format Control" in the Embedded Coder documentation

Generate scalar inlined parameter as

Control expression of scalar inlined parameter values in the generated code.

Settings

Default: Literals

Literals

Generates scalar inlined parameters as numeric constants. This setting can help with debugging TLC code, as it makes it easy to search for parameter values in the generated code.

Macros

Generates scalar inlined parameters as variables with #define macros. This setting makes generated code more readable.

Dependencies

- This parameter appears only for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `InlinedPrmAccess`
Type: string
Value: `Literals` | `Macros`
Default: `Literals`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Macros`
Efficiency	`Literals`
Safety precaution	No impact

Signal naming

Specify rules for naming signals in generated code.

Settings

Default: None

None

   Does not change signal names when creating corresponding identifiers in generated code. Signal identifiers in the generated code match the signal names that appear in the model.

Force upper case

   Uses uppercase characters when creating identifiers for signal names in the generated code.

Force lower case

   Uses lowercase characters when creating identifiers for signal names in the generated code.

Custom M-function

Uses the MATLAB function specified with the **M-function** parameter to create identifiers for signal names in the generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.
- If you give a value to the **Alias** parameter of an `MPT.Signal` or `Simulink.Signal` data object, that value overrides the specification of the **Signal naming** parameter.

Limitation

This parameter does not impact signal names that are specified by an embedded signal object created using the **Code Generation** tab of a **Signal Properties** dialog box. See "Custom Storage Classes Using Embedded Signal Objects" for information about embedded signal objects.

Command-Line Information

Parameter: `SignalNamingRule`
Type: string
Value: `None` | `UpperCase` | `LowerCase` | `Custom`
Default: `None`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Force upper case`
Efficiency	No impact
Safety precaution	No impact

See Also

- "Apply Naming Rules to Identifiers Globally" in the Embedded Coder documentation
- "Programming Scripts and Functions" in the MATLAB documentation

M-function

Specify rule for naming identifiers in generated code.

Settings

Default: ' '

Enter the name of a MATLAB language file that contains the naming rule to be applied to signal, parameter, or #define parameter identifiers in generated code. Examples of rules you might program in such a MATLAB function include:

- Remove underscore characters from signal names.
- Add an underscore before uppercase characters in parameter names.
- Make identifiers uppercase in generated code.

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by **Signal naming**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information
Parameter: SignalNamingFcn
Type: string
Value: MATLAB language file
Default: ' '

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Apply Naming Rules to Identifiers Globally" in the Embedded Coder documentation
- "Programming Scripts and Functions" in the MATLAB documentation

Parameter naming

Specify rule for naming parameters in generated code.

Settings

Default: None

None

> Does not change parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case

> Uses uppercase characters when creating identifiers for parameter names in the generated code.

Force lower case

> Uses lowercase characters when creating identifiers for parameter names in the generated code.

Custom M-function

> Uses the MATLAB function specified with the **M-function** parameter to create identifiers for parameter names in the generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to Custom M-function enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: ParamNamingRule
Type: string
Value: None | UpperCase | LowerCase | Custom

Default: None

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Force upper case
Efficiency	No impact
Safety precaution	No impact

See Also

- "Apply Naming Rules to Identifiers Globally" in the Embedded Coder documentation
- "Programming Scripts and Functions" in the MATLAB documentation

#define naming

Specify rule for naming #define parameters (defined with storage class Define (Custom)) in generated code.

Settings

Default: None

None

   Does not change #define parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case

   Uses uppercase characters when creating identifiers for #define parameter names in the generated code.

Force lower case

   Uses lowercase characters when creating identifiers for #define parameter names in the generated code.

Custom M-function

   Uses the MATLAB function specified with the **M-function** parameter to create identifiers for #define parameter names in the generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information
Parameter: `DefineNamingRule`
Type: string
Value: `None` | `UpperCase` | `LowerCase` | `Custom`
Default: `None`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Force upper case`
Efficiency	No impact
Safety precaution	No impact

See Also

- "Apply Naming Rules to Identifiers Globally" in the Embedded Coder documentation
- "Programming Scripts and Functions" in the MATLAB documentation

Use the same reserved names as Simulation Target

Specify whether to use the same reserved names as those specified in the **Simulation Target** > **Symbols** pane.

Settings

Default: Off

☑ On

 Enables using the same reserved names as those specified in the **Simulation Target** > **Symbols** pane.

4-95

☐ Off

> Disables using the same reserved names as those specified in the **Simulation Target** > **Symbols** pane.

Command-Line Information
Parameter: UseSimReservedNames
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Reserved names

Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

Settings

Default: {}

This action changes the names of variables or functions in the generated code to avoid name conflicts with identifiers in custom code. Reserved names must be shorter than 256 characters.

Tips

- Do not enter Simulink Coder keywords since these names cannot be changed in the generated code. For a list of keywords to avoid, see "Reserved Keywords".
- Start each reserved name with a letter or an underscore to prevent error messages.
- Each reserved name must contain only letters, numbers, or underscores.
- Separate the reserved names using commas or spaces.

- You can also specify reserved names by using the command line:

  ```
  config_param_object.set_param('ReservedNameArray', {'abc','xyz'})
  ```

  where *config_param_object* is the object handle to the model settings in the Configuration Parameters dialog box.

Command-Line Information
Parameter: ReservedNameArray
Type: string array
Value: reserved names shorter than 256 characters
Default: {}

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Code Generation Pane: Custom Code

The **Code Generation** > **Custom Code** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT- or ERT-based target.

☐ Use the same custom code settings as Simulation Target

Include custom C code in generated:

Source file
Header file
Initialize function
Terminate function

Source file:

Include list of additional:

Include directories
Source files
Libraries

Include directories:

Code Generation: Custom Code Tab Overview

Enter custom code to include in generated model files and create a list of additional folders, source files, and libraries to use when building the model.

Configuration

1   Select the type of information to include from the list on the left side of the pane.
2   Enter custom code or enter a string to identify a folder, source file, or library.
3   Click **Apply**.

See Also

- "Configure Model for External Code Integration"
- "Code Generation Pane: Custom Code" on page 4-98

Use the same custom code settings as Simulation Target

Specify whether to use the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.

Settings

Default: Off

☑ On

Enables using the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.

☐ Off

Disables using the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.

Command-Line Information

Parameter: `RTWUseSimCustomCode`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Use local custom code settings (do not inherit from main model)

Specify if a library model can use custom code settings that are unique from the main model.

Settings

Default: Off

☑ On

Enables a library model to use custom code settings that are unique from the main model.

☐ Off

> Disables a library model from using custom code settings that are unique from the main model.

Dependency

This parameter is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks.

Command-Line Information

Parameter: `RTWUseLocalCustomCode`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Source file

Specify custom code to include near the top of the generated model source file.

Settings

Default: `' '`

The code generator places code near the top of the generated *model*.c or *model*.cpp file, outside of any function.

Command-Line Information

Parameter: `CustomSourceCode`

Type: string
Value: C code
Default: ' '

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Header file

Specify custom code to include near the top of the generated model header file.

Settings

Default: ' '

The Simulink Coder software places this code near the top of the generated *model*.h
header file. If you are including a header file, in your custom header file add #ifndef
code. This avoids multiple inclusions. For example, in rtwtypes.h the following
#include guards are added:

```
#ifndef RTW_HEADER_rtwtypes_h_
#define RTW_HEADER_rtwtypes_h_
...
#endif /* RTW_HEADER_rtwtypes_h_ */
```

Command-Line Information
Parameter: CustomHeaderCode
Type: string
Value: C code
Default: ' '

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Initialize function

Specify custom code to include in the generated model initialize function.

Settings

Default: ' '

The Simulink Coder software places code inside the model's initialize function in the *model*.c or *model*.cpp file.

Command-Line Information
Parameter: `CustomInitializer`
Type: string
Value: C code
Default: ' '

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Terminate function

Specify custom code to include in the generated model terminate function.

Settings

Default: ' '

Specify code to appear in the model's generated terminate function in the *model*.c or *model*.cpp file.

Dependency

A terminate function is generated only if you select the **Terminate function required** check box on the **Code Generation > Interface** pane.

Command-Line Information
Parameter: CustomTerminator
Type: string
Value: C code
Default: ' '

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Include directories

Specify a list of include folders to add to the include path.

Settings

Default:`''`

Enter a space-separated list of include folders to add to the include path when compiling the generated code.

- Specify absolute or relative paths to the folders.
- Relative paths must be relative to the folder containing your model files, not relative to the build folder.
- The order in which you specify the folders is the order in which they are searched for header, source, and library files.

Note: If you specify a Windows path string containing one or more spaces, you must enclose the string in double quotes. For example, the second and third path strings in the **Include directories** entry below must be double-quoted:

```
C:\Project "C:\Custom Files" "C:\Library Files"
```

If you set the equivalent command-line parameter `CustomInclude`, each path string containing spaces must be separately double-quoted within the single-quoted third argument string, for example,

```
>> set_param('mymodel', 'CustomInclude', ...
            'C:\Project "C:\Custom Files" "C:\Library Files"')
```

Command-Line Information
Parameter: `CustomInclude`
Type: string
Value: folder path
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Source files

Specify a list of additional source files to compile and link with the generated code.

Settings

Default: ' '

Enter a space-separated list of source files to compile and link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

You can specify just the file name if the file is in the current MATLAB folder or in one of the include folders.

Command-Line Information
Parameter: `CustomSource`
Type: string
Value: file name
Default: ' '

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

4-107

Application	Setting
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Libraries

Specify a list of additional libraries to link with the generated code.

Settings

Default: ' '

Enter a space-separated list of static library files to link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

You can specify just the file name if the file is in the current MATLAB folder or in one of the include folders.

Command-Line Information
Parameter: `CustomLibrary`
Type: string
Value: library file name
Default: ' '

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure Model for External Code Integration"

Code Generation Pane: Debug

The **Code Generation** > **Debug** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT- or ERT-based target.

Build process
☑ Verbose build
☐ Retain .rtw file

TLC process
☐ Profile TLC
☐ Start TLC debugger when generating code
☐ Start TLC coverage when generating code
☐ Enable TLC assertion

In this section...
"Code Generation: Debug Tab Overview" on page 4-110
"Verbose build" on page 4-111
"Retain .rtw file" on page 4-111
"Profile TLC" on page 4-112
"Start TLC debugger when generating code" on page 4-113
"Start TLC coverage when generating code" on page 4-114
"Enable TLC assertion" on page 4-115

Code Generation: Debug Tab Overview

Select build process and Target Language Compiler (TLC) process options.

See Also

- "Debug"
- "Code Generation Pane: Debug" on page 4-110

Verbose build

Display code generation progress.

Settings

Default: on

☑ On

> The MATLAB Command Window displays progress information indicating code generation stages and compiler output during code generation.

☐ Off

> Does not display progress information.

Command-Line Information
Parameter: `RTWVerbose`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	On

See Also

"Debug"

Retain .rtw file

Specify *model*`.rtw` file retention.

Settings

Default: off

☑ On

> Retains the *model*.rtw file in the current build folder. This parameter is useful if you are modifying the target files and need to look at the file.

☐ Off

> Deletes the *model*.rtw from the build folder at the end of the build process.

Command-Line Information
Parameter: RetainRTWFile
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Debug"

Profile TLC

Profile the execution time of TLC files.

Settings

Default: off

☑ On

> The TLC profiler analyzes the performance of TLC code executed during code generation, and generates an HTML report.

☐ Off

    Does not profile the performance.

Command-Line Information
Parameter: `ProfileTLC`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Debug"

Start TLC debugger when generating code

Specify use of the TLC debugger

Settings

Default: Off

☑ On

    The TLC debugger starts during code generation.

☐ Off

    Does not start the TLC debugger.

Tips

• You can also start the TLC debugger by entering the `-dc` argument into the **System target file** field.

- To invoke the debugger and run a debugger script, enter the -df *filename* argument into the **System target file** field.

Command-Line Information
Parameter: TLCDebug
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Debug"

Start TLC coverage when generating code

Generate the TLC execution report.

Settings

Default: off

☑ On

   Generates .log files containing the number of times each line of TLC code is executed during code generation.

☐ Off

   Does not generate a report.

Tip

You can also generate the TLC execution report by entering the -dg argument into the **System target file** field.

Command-Line Information

Parameter: `TLCCoverage`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Debug"

Enable TLC assertion

Produce the TLC stack trace

Settings

Default: off

☑ On

> The build process halts if a user-supplied TLC file contains an `%assert` directive that evaluates to FALSE.

☐ Off

> The build process ignores TLC assertion code.

Command-Line Information

Parameter: `TLCAssert`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	On

See Also

"Debug"

Code Generation Pane: Interface

The **Code Generation** > **Interface** pane includes the following parameters when the Simulink Coder product is installed on your system and you select a GRT-based target.

The **Code Generation** > **Interface** pane includes additional parameters when the Simulink Coder product is installed on your system and you select an ERT-based target. ERT-based target parameters require an Embedded Coder license when generating code.

In this section...
"Code Generation: Interface Tab Overview" on page 4-120
"Standard math library" on page 4-120
"Code replacement library" on page 4-121
"Custom" on page 4-124
"Shared code placement" on page 4-124
"Support: floating-point numbers" on page 4-125
"Support: non-finite numbers" on page 4-126
"Support: complex numbers" on page 4-127
"Support: absolute time" on page 4-128

Code Generation: Interface Tab Overview

Select the target software environment, output variable name modifier, and data exchange interface.

See Also

- "Specify Target Interfaces"
- "Code Generation Pane: Interface" on page 4-117

Standard math library

Specify a standard math library for your model.

Settings

Default: C89/C90 (ANSI)

C89/C90 (ANSI)

   Generates calls to the ISO®/IEC 9899:1990 C standard math library for floating-point functions.

C99 (ISO)

   Generates calls to the ISO/IEC 9899:1999 C standard math library.

C++03 (ISO)

   Generates calls to the ISO/IEC 14882:2003 C++ standard math library.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Dependencies

The C++03 (ISO) math library is available for use only if you select C++ for the **Language** parameter.

Command-Line Information
Parameter: TargetLangStandard
Type: string
Value: 'C89/C90 (ANSI)' | 'C99 (ISO)' | 'C++03 (ISO)'
Default: 'C89/C90 (ANSI)'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Specify Target Interfaces"

Code replacement library

Specify an application-specific math library for your model.

Settings

Default: None

None
    Does not use a code replacement library.
GNU C99 extensions

Generates calls to the GNU® gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

`Intel IPP for x86-64 (Windows)`

Generates calls to the Intel® Performance Primitives (IPP) library for the x86-64 Windows platform.

`Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)`

Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Windows platform.

`Intel IPP for x86/Pentium (Windows)`

Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.

`Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)`

Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform.

`Intel IPP for x86-64 (Linux)`

Generates calls to the Intel Performance Primitives (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 Intel Performance Primitives (IPP) and Streaming SIMD Extensions (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 the **Language**, **Standard math library**, and **Device vendor** values you select for your model.

Tips

- If you specify `Shared location` for the **Code Generation** > **Interface** > **Shared code placement** parameter or you generate code for models in a model reference hierarchy,

- Models that are sharing the location or are in the model hierarchy must specify the same code replacement library (same name, tables, and table entries).

- The code generator reports a checksum warning (see "Shared Utility Checksum") if you change the name or contents of the code replacement library and rebuild the model from the same folder as the previous build. The warning prompts you to remove the existing folder and stop or stop code generation.

- If both of the following conditions exist for a model that contains Stateflow charts, the Simulink software regenerates code for the charts and recompiles the generated code.

  - You *do not* specify `Shared location` for the **Code Generation** > **Interface** > **Shared code placement** parameter.

  - You change the code replacement library name or contents before regenerating code.

Tip

Before setting this parameter, verify that your compiler supports the library that you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Command-Line Information
Parameter: `CodeReplacementLibrary`
Type: string
Value: `'None'` | `'GNU C99 extensions'` | `'Intel IPP'` | `'Intel IPP/SSE with GNU99 extensions'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Specify Target Interfaces"

Custom

Open the Code Replacement Tool. With this tool, you can you create and manage the code replacement tables that make up a code replacement library (CRL).

Dependencies

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

See Also

- "What Is Code Replacement Customization?""Develop a Code Replacement Library"

Shared code placement

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Settings

Default: `Auto`

`Auto`

Operates as follows:

- When the model contains Model blocks, places utility code within the `slprj/target/_sharedutils` folder.
- When the model does not contain Model blocks, places utility code in the build folder (generally, in *model*`.c` or *model*`.cpp`).

`Shared location`

Directs code for utilities to be placed within the `slprj` folder in your working folder.

Command-Line Information
Parameter: `UtilityFuncGeneration`
Type: string
Value: `'Auto'` | `'Shared location'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	`Shared location` (GRT) No impact (ERT)
Traceability	`Shared location` (GRT) No impact (ERT)
Efficiency	No impact (execution, RAM) `Shared location` (ROM)
Safety precaution	No impact

See Also

- "Specify Target Interfaces"
- "Sharing Utility Code"

Support: floating-point numbers

Specify whether to generate floating-point data and operations.

Settings

Default: On (GUI), `'off'` (command-line)

☑ On

Generates floating-point data and operations.

☐ Off

Generates pure integer code. If you clear this option, an error occurs if the code generator encounters floating-point data or expressions. The error message reports offending blocks and parameters.

Dependencies

- This option only appears for ERT-based targets.
- This option requires an Embedded Coder license when generating code.
- Selecting this option enables **Support: non-finite numbers** and clearing this option disables **Support: non-finite numbers**.
- This option must be the same for top-level and referenced models.

Command-Line Information
Parameter: `PurelyIntegerCode`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Note: The command-line values are reverse of the settings values. Therefore, `'on'` in the command line corresponds to the description of "Off" in the settings section, and `'off'` in the command line corresponds to the description of "On" in the settings section.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (GUI), `'on'` (command-line) — for integer only
Safety precaution	No impact

Support: non-finite numbers

Specify whether to generate nonfinite data and operations on nonfinite data.

Settings

Default: on

☑ On

Generates nonfinite data (for example, `NaN` and `Inf`) and related operations.

☐ Off

Does not generate nonfinite data and operations. If you clear this option, an error occurs if the code generator encounters nonfinite data or expressions. The error message reports offending blocks and parameters.

Note: Code generation is optimized with the assumption that nonfinite data are absent. However, if your application produces nonfinite numbers through signal data

---

or MATLAB code, the behavior of the generated code might be inconsistent with simulation results when processing nonfinite data.

Dependencies

- For ERT-based targets, this parameter is enabled by **Support: floating-point numbers**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: SupportNonFinite
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

Support: complex numbers

Specify whether to generate complex data and operations.

Settings

Default: on

☑ On

Generates complex numbers and related operations.

☐ Off

Does not generate complex data and related operations. If you clear this option, an error occurs if the code generator encounters complex data or expressions. The error message reports offending blocks and parameters.

4-127

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `SupportComplex`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (for real only)
Safety precaution	No impact

Support: absolute time

Specify whether to generate and maintain integer counters for absolute and elapsed time values.

Settings

Default: on

☑ On

  Generates and maintains integer counters for blocks that require absolute or elapsed time values. Absolute time is the time from the start of program execution to the present time. An example of elapsed time is time elapsed between two trigger events.

  If you select this option and the model does not include blocks that use time values, the target does not generate the counters.

☐ Off

Does not generate integer counters to represent absolute or elapsed time values. If you do not select this option and the model includes blocks that require absolute or elapsed time values, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- You must select this parameter if your model includes blocks that require absolute or elapsed time values.

Command-Line Information

Parameter: SupportAbsoluteTime
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

"Timers"

Support: continuous time

Specify whether to generate code for blocks that use continuous time.

Settings

Default: off

☑ On

Generates code for blocks that use continuous time.

☐ Off

> Does not generate code for blocks that use continuous time. If you do not select this option and the model includes blocks that use continuous time, an error occurs during code generation.

Dependencies

- This option only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This option must be on if your model includes blocks that require absolute or elapsed time values.
- If you have customized `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, be aware that selecting the options **Support: continuous time** and **Single output/update function** together may cause output values read from `ert_main` for a continuous output port to differ from the corresponding output values in the logged data for the model. This is because, while logged data is a snapshot of output at major time steps, output read from `ert_main` after the base-rate model step potentially reflects intervening minor time steps. To work around this limitation and eliminate the discrepancy, do one of the following:

  - Separate the generated output and update functions (clear the **Single output/ update function** option), and insert code in `ert_main` to read model output values reflecting only the major time steps. For example, in `ert_main`, between the *model_output* call and the *model_update* call, read the model `External outputs` global data structure (defined in *model*.h).

  - If you want to keep the **Single output/update function** option selected, insert code in the generated *model*.c or `.cpp` file to return model output values reflecting only the major time steps. For example, in the model step function, between the output code and the update code, you could save the value of the model `External outputs` global data structure (defined in *model*.h), and then restore the value after the update code completes.

  - Place a Zero-Order Hold block before the continuous output port.

Command-Line Information
Parameter: `SupportContinuousTime`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

"Use Discrete and Continuous Time "

Support: non-inlined S-functions

Specify whether to generate code for noninlined S-functions.

Settings

Default: Off

☑ On

 Generates code for noninlined S-functions.

☐ Off

 Does not generate code for noninlined S-functions. If this parameter is off and the
 model includes a noninlined S-function, an error occurs during the build process.

Tip

- Inlining S-functions is highly advantageous in production code generation, for
 example, for implementing device drivers. In such cases, clear this option to enforce
 use of inlined S-functions for code generation.
- Noninlined S-functions require additional memory and computation resources, and
 can result in significant performance issues. Consider using an inlined S-function
 when efficiency is a concern.

Dependencies

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

- Selecting this parameter also selects **Support: floating-point numbers** and **Support: non-finite numbers**. If you clear **Support: floating-point numbers** or **Support: non-finite numbers**, a warning is displayed during code generation because these parameters are required by the S-function interface.

Command-Line Information

Parameter: `SupportNonInlinedSFcns`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

"Insert S-Function Code"

Support: variable-size signals

Specify whether to generate code for models that use variable-size signals.

Settings

Default: Off

☑ On

Generates code for models that use variable-size signals.

☐ Off

Does not generate code for models that use variable-size signals. If this parameter is off and the model uses variable-size signals, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `SupportVariableSizeSignals`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off
Safety precaution	Off

Multiword type definitions

Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.

Settings

Default: `System defined`

`System defined`

Use the default system type definitions for multiword data types in generated code. During code generation, if multiword usage is detected, multiword type definitions are generated into the file `multiword_types.h`.

`User defined`

Allows you to control how multiword type definitions are handled during the code generation process. Selecting this value enables the associated parameter **Maximum word length**, which allows you to specify a maximum word length, in bits, for which the code generation process generates multiword type definitions into the file `multiword_types.h`. The default maximum word length is 256. If you select 0, multiword type definitions are not generated into the file `multiword_types.h`.

The maximum word length for multiword types only determines the type definitions generated and does not impact the efficiency of the generated code. If the maximum word length for multiword types is set to 0 or too small, an error occurs when the

generated code is compiled. This error is caused by the generated code using a type that does not have the required type definition. To resolve the error, increase the maximum word length and regenerate the code. If the maximum word length for multiword types is larger than required, then `multiword_types.h` might contain unused type definitions. Unused type definitions do not consume target resources.

Tips

- Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `multiword_types.h` file during code generation. These updates occur when the new model uses multiword types of length greater than those of the other models. You must then recompile and, depending on your development process, reverify previously generated code. To prevent updates to `multiword_types.h`, determine a maximum word length sufficiently big to cover the needs of all models in the hierarchy. Configure every model in the hierarchy to use that same maximum word length.

- The majority of embedded designs do not need multiword types. By setting maximum word length for multiword types to 0, you can prevent use of multiword variables on the target. If you use multiword variables with a maximum word length that is 0 or smaller than required, you are alerted with an error when the generated code is compiled.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Selecting the value `User defined` for this parameter enables the associated parameter **Maximum word length**.

Command-Line Information

Parameter: ERTMultiwordTypeDef
Type: string
Value: `'System defined'` | `'User defined'`
Default: `'System defined'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	Use default

Maximum word length

Specify a maximum word length, in bits, for which the code generation process generates system-defined multiword type definitions.

Settings

Default: 256

Specify a maximum word length, in bits, for which the code generation process generates multiword type definitions into the file `multiword_types.h`. All multiword type definitions up to and including this number of bits are generated. If you select 0, multiword type definitions are not generated into the file `multiword_types.h`.

The maximum word length for multiword types only determines the type definitions generated and does not impact the efficiency of the generated code. If the maximum word length for multiword types is set to 0 or too small, an error occurs when the generated code is compiled. This error is caused by the generated code using a type that does not have the required type definition. To resolve the error, increase the maximum word length and regenerate the code. If the maximum word length for multiword types is larger than required, then `multiword_types.h` might contain unused type definitions. Unused type definitions do not consume target resources.

Tips

- Adding a model to a model hierarchy or changing an existing model in the hierarchy can result in updates to the shared `multiword_types.h` file during code generation. These updates occur when the new model uses multiword types of length greater than those of the other models. You must then recompile and, depending on your development process, reverify previously generated code. To prevent updates to `multiword_types.h`, determine a maximum word length sufficiently big to cover the needs of all models in the hierarchy. Configure every model in the hierarchy to use that same maximum word length.

- The majority of embedded designs do not need multiword types. By setting maximum word length for multiword types to 0, you can prevent use of multiword variables on the target. If you use multiword variables with a maximum word length that is 0

or smaller than required, you are alerted with an error when the generated code is compiled.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by selecting the value `User defined` for the parameter **Multiword type definitions**.

Command-Line Information
Parameter: ERTMultiwordLength
Type: integer
Value: valid quantity of bits representing a word size
Default: 256

Recommended Settings

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

Code interface packaging

Select the packaging for the generated C or C++ code interface.

Settings

Default: `Nonreusable function` if **Language** is set to `C`; `C++ class` if **Language** is set to `C++`

`C++ class`

Generate a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.

`Nonreusable function`

Generate nonreusable code. Model data structures are statically allocated and accessed by model entry point functions directly in the model code.

Reusable function

Generate reusable, multi-instance code that is reentrant, as follows:

- For a GRT-based model, the generated *model*.c source file contains an allocation function that dynamically allocates model data for each instance of the model. For an ERT-based model, you can use the **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 hand-written code, add an #include model.h directive to the code.

- When Reusable function is selected, the code generator generates a pointer to the real-time model object (*model*_M).

- In some cases, when Reusable function is selected, the code generator might generate code that compiles but is not reentrant. For example, if a signal, DWork structure, or parameter data has a storage class other than Auto, global data structures are generated.

Dependencies

- The value C++ class is available only if the **Language** parameter is set to C++ on the **Code Generation** pane.

- Selecting Reusable function or C++ class enables **Multi-instance code error diagnostic**.

- For an ERT target, selecting Reusable function enables **Pass root-level I/O as** and **Use dynamic memory allocation for model initialization**.

- For an ERT target, selecting C++ class enables the following controls for customizing the model class interface:

4-137

- **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, provided that 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 the same source
    - Has a port that is used by multiple instances of the subsystem and has different sample times, data types, complexity, frame status, or dimensions across the instances
    - Has output marked as a global signal
    - For each instance contains identical blocks with different names or parameter settings

- Using `Reusable function` does not impact the code generated for function-call subsystems.

Command-Line Information
Parameter: `CodeInterfacePackaging`
Type: string
Value: `'C++ class'` | `'Nonreusable function'` | `'Reusable function'`
Default: `'Nonreusable function'` if `TargetLang` is set to `'C'`; `'C++ class'` if `TargetLang` is set to `'C++'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	`Reusable function` or `C++ class`
Safety precaution	No impact

See Also

- "Entry-Point Functions and Scheduling"
- "Generate Reentrant Code from Top-Level Models" (GRT)
- "Use GRT with Reusable Function Packaging to Combine Models"
- "Generate Reentrant Code from Top-Level Models" (ERT)
- "Generate C++ Class Interface to Model or Subsystem Code" (GRT)
- "C++ Class Interface Control" (ERT)
- "Code Generation of Subsystems"
- "Code Reuse Limitations for Subsystems"
- "Determine Why Subsystem Code Is Not Reused"
- "S-Functions That Support Code Reuse"
- "Static Main Program Module"
- "Function Prototype Control"
- "Atomic Subsystem Code"
- "Export Function-Call Subsystems"
- `model_step`

Multi-instance code error diagnostic

Select the severity level for diagnostics displayed when a model violates requirements for generating multi-instance code.

Settings

Default: `Error`

`None`

   Proceed with build without displaying a diagnostic message.

`Warning`

Proceed with build after displaying a warning message.

`Error`

Abort build after displaying an error message.

Under certain conditions, the software might

- Generate code that compiles but is not reentrant. For example, if 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 coder generates invalid code, displays an error message, and terminates the build.

Dependencies

This parameter is enabled by setting **Code interface packaging** to `Reusable function` or `C++ class`.

Command-Line Information

Parameter: `MultiInstanceErrorCode`
Type: string
Value: `'None'` | `'Warning'` | `'Error'`
Default: `'Error'`

Recommended Settings

Application	Setting
Debugging	`Warning` or `Error`
Traceability	No impact
Efficiency	`None`
Safety precaution	No impact

See Also

- "Entry-Point Functions and Scheduling"
- "Generate Reentrant Code from Top-Level Models"
- "Generate C++ Class Interface to Model or Subsystem Code"
- "Code Generation of Subsystems"
- "Code Reuse Limitations for Subsystems"

- "Determine Why Subsystem Code Is Not Reused"
- "Atomic Subsystem Code"

Pass root-level I/O as

Control how root-level model input and output are passed to the reusable *model*_step function.

Settings

Default: `Individual arguments`

`Individual arguments`

> Passes each root-level model input and output value to *model*_step as a separate argument.

`Structure reference`

> Packs root-level model input into a `struct` and passes `struct` to *model*_step as an argument. Similarly, packs root-level model output into a second `struct` and passes it to *model*_step.

`Part of model data structure`

> Packages root-level model input and output into the real-time model data structure.

Dependencies

- This parameter only appears for ERT-based targets with **Code interface packaging** set to `Resuable function`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `RootIOFormat`
Type: string
Value: `'Individual arguments'` | `'Structure reference'` | `'Part of model data structure'`
Default: `'Individual arguments'`

Recommended Settings

Application	Setting
Debugging	No impact

4-141

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Entry-Point Functions and Scheduling"
- "Generate Reentrant Code from Top-Level Models"
- "Code Generation of Subsystems"
- "Atomic Subsystem Code"
- model_step

Classic call interface

Specify whether to generate model function calls compatible with the main program module of the GRT target in models created before R2012a.

Settings

Default: off (except on for GRT models created before R2012a)

☑ On

Generates model function calls that are compatible with the main program module of the GRT target (grt_main.c or grt_main.cpp) in models created before R2012a.

This option provides a quick way to use code generated in the current release with a GRT-based custom target that has a main program module based on pre-R2012a grt_main.c or grt_main.cpp.

☐ Off

Disables the classic call interface.

Tips

The following are unsupported:

- Data type replacement
- Nonvirtual subsystem option **Function with separate data**

Dependencies

- 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: string
Value: 'on' | 'off'
Default: 'off' (except 'on' for GRT models created before R2012a)

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

"Use Discrete and Continuous Time "

Use dynamic memory allocation for model initialization

Control how the generated code allocates memory for model data.

Settings

Default: off

☑ On

Generates a function to dynamically allocate memory (using malloc) for model data structures.

☐ Off

> Does not generate a dynamic memory allocation function. The generated code statically allocates memory for model data structures.

Dependencies

- This parameter only appears for ERT-based targets with **Code interface packaging** set to `Resuable function`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `GenerateAllocFcn`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	Off

See Also

- "Entry-Point Functions and Scheduling"
- "Generate Reentrant Code from Top-Level Models"
- "Code Generation of Subsystems"
- "Atomic Subsystem Code"
- `model_step`

Use dynamic memory allocation for model block instantiation

Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ class interface.

Settings

Default: off

☑ On

> Generates code that uses dynamic memory allocation to instantiate objects for referenced models configured with a C++ class interface. Specifically, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses `new` to instantiate objects for referenced models.

> Selecting this option frees a parent model from having to maintain information about referenced models beyond its direct children.

> - If you select this option, be aware that a `bad_alloc` exception might be thrown, per the C++ standard, if an out-of-memory error occurs during the use of `new`. You must provide code to catch and process the `bad_alloc` exception in case an out-of-memory error occurs for a `new` call during construction of a top model object.
> - If **Use 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: string
Value: `'on'` | `'off'`

Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	Off

See Also

"Configure Code Interface Options"

Single output/update function

Specify whether to generate the *model*_step function.

Settings

Default: on

☑ On

Generates the *model*_step function for a model. This function contains the output and update function code for the blocks in the model and is called by rt_OneStep to execute processing for one clock period of the model at interrupt level.

☐ Off

Does not combine output and update function code into a single function, and instead generates the code in separate *model*_output and *model*_update functions.

Tips

Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. See "Model Blocks and Direct Feed through" for details.

Simulink Coder ignores this parameter for a referenced model if any of the following conditions apply to that model:

- Is multi-rate
- Has a continuous sample time
- Is logging states (using the **States** or **Final states** parameters in the **Configuration Parameters** > **Data Import/Export** pane

Dependencies

- 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 have customized `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, be aware that selecting the options **Support: continuous time** and **Single output/update function** together may cause output values read from `ert_main` for a continuous output port to differ from the corresponding output values in the logged data for the model. This is because, while logged data is a snapshot of output at major time steps, output read from `ert_main` after the base-rate model step potentially reflects intervening minor time steps. To work around this limitation and eliminate the discrepancy, do one of the following:

  - Separate the generated output and update functions (clear the **Single output/ update function** option), and insert code in `ert_main` to read model output values reflecting only the major time steps. For example, in `ert_main`, between the *model*`_output` call and the *model*`_update` call, read the model `External outputs` global data structure (defined in *model*`.h`).
  - If you want to keep the **Single output/update function** option selected, insert code in the generated *model*`.c` or `.cpp` file to return model output values reflecting only the major time steps. For example, in the model step function, between the output code and the update code, you could save the value of the model `External outputs` global data structure (defined in *model*`.h`), and then restore the value after the update code completes.
  - Place a Zero-Order Hold block before the continuous output port.

Command-Line Information
Parameter: `CombineOutputUpdateFcns`
Type: string

Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	On
Safety precaution	On

See Also

"rt_OneStep and Scheduling Considerations"

Terminate function required

Specify whether to generate the *model*_`terminate` function.

Settings

Default: on

☑ On

> Generates a *model*_`terminate` function. This function contains model termination code and should be called as part of system shutdown.

☐ Off

> Does not generate a *model*_`terminate` function. Suppresses the generation of this function if you designed your application to run indefinitely and does not require a terminate function.

Dependencies

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

Command-Line Information
Parameter: `IncludeMdlTerminateFcn`

Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

`model_terminate`

Generate preprocessor conditionals

Generate preprocessor conditional directives globally for a model or locally for each Model block with variant models.

Settings

Default: `Use local settings`

`Use local settings`

 Generates preprocessor conditional directives based on the value of the **Generate preprocessor conditionals** parameter on the Model block parameters dialog. If you select the **Generate preprocessor conditionals** parameter in the Model block parameters dialog, the generated code contains preprocessor conditional directives for all variant models of that Model block. If you do not select this parameter for a Model block, code is generated for the active variant model.

`Enable all`

 Generates preprocessor conditional directives for all variant models of the Model blocks. Disables the **Generate preprocessor conditionals** option in the Model block parameters dialog.

`Disable all`

Only generates code for the active variant model of the Model block. Disables the **Generate preprocessor conditionals** option in the Model block parameters dialog for Model blocks.

Tips

For generating preprocessor directives we recommend the following settings:

- Select the "Inline parameters" parameter on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box.
- Deselect the "Ignore custom storage classes" parameter on the **Code Generation** pane of the Configuration Parameters dialog box.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Use local settings` enables **Generate preprocessor conditionals** parameter on the Model block parameters dialog.
- Setting this parameter to `Enable all` or `Disable all` disables the **Generate preprocessor conditionals** check box on the Model block parameters dialog.
- Setting this parameter to `Enable all` sets the **Selected variant** control on the Model block parameter dialog to `(derive from conditions)`.

Command-Line Information
Parameter: `GeneratePreprocessorConditionals`
Type: string
Value: `'Use local settings'` | `'Enable all'` | `'Disable all'`
Default: `'Use local settings'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Workflow for Implementing Variants"
- "Variant Systems"

Suppress error status in real-time model data structure

Specify whether to log or monitor error status.

Settings

Default: off

☑ On

> Omits the error status field from the generated real-time model data structure `rtModel`. This option reduces memory usage.
>
> Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

☐ Off

> Includes an error status field in the generated real-time model data structure `rtModel`. You can use available macros to monitor the field for error message data or set it with error message data.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is cleared if you select the incompatible option **MAT-file logging**. If you subsequently select this parameter, code generation displays an error.
- Selecting this parameter clears **Support: continuous time**.
- If your application contains multiple integrated models, the setting of this option must be the same for all of the models to avoid unexpected application behavior. For example, if you select the option for one model but not another, an error status might not get registered by the integrated application.

Command-Line Information
Parameter: `SuppressErrorStatus`

4-151

Type: string
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact
Efficiency	On
Safety precaution	On

See Also

"Use the Real-Time Model Data Structure"

Combine signal/state structures

Specify whether to combine global block signals and global state data into one data structure in the generated code

Settings

Default: Off

☑ On

Combine global block signal data (block I/O) and global state data (DWork vectors) into one data structure in the generated code.

☐ Off

Store global block signals and global states in separate data structures, block I/O and DWork vectors, in the generated code.

Tips

The benefits to setting this parameter to On are:

- Enables tighter memory representation through fewer bitfields, which reduces RAM usage
- Enables better alignment of data structure elements, which reduces RAM usage

- Reduces the number of arguments to reusable subsystem and model reference block functions, which reduces stack usage
- Better readable data structures with more consistent element sorting

Example

For a model that generates the following code:

```
/* Block signals (auto storage) */
typedef struct {
  struct {
    uint_T LogicalOperator:1;
    uint_T UnitDelay1:1;
  } bitsForTID0;
} BlockIO;
/* Block states (auto storage) */
typedef struct {
  struct {
    uint_T UnitDelay_DSTATE:1
    uint_T UnitDelay1_DSTATE:1
  } bitsForTID0;
} D_Work;
```

If you select **Combine signal/state structures**, the generated code now looks like this:

```
/* Block signals and states (auto storage)
   for system */
typedef struct {
  struct {
    uint_T LogicalOperator:1;
    uint_T UnitDelay1:1;
    uint_T UnitDelay_DSTATE:1;
    uint_T UnitDelay1_DSTATE:1;
  } bitsForTID0;
} D_Work;
```

Dependencies

This parameter:

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

Command-Line Information
Parameter: `CombineSignalStateStructs`

4-153

Type: string
Value: `'on'` | `'off'`
Default: `off`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On
Safety precaution	No impact

See Also

- "Global Block I/O Structure"
- "State Storage"

Configure Model Functions

Open the Model Interface dialog box. In this dialog box, you can specify whether the code generator uses default *model*_`initialize` and *model*_`step` function prototypes or model-specific C prototypes. Based on your selection, you can preview and modify the function prototypes.

Dependencies

- This button appears only for ERT-based targets with **Code interface packaging** set to a value other than `C++ class`.
- This button requires an Embedded Coder license when generating code.
- This button is active only if your model uses an attached configuration set. If your model uses a referenced configuration set, the button is greyed out. If you want to configure a model-specific step function prototype for a referenced configuration set, use the MATLAB function prototype control functions described in "Configure Function Prototypes Programmatically".

See Also

- "Function Prototype Control"
- `model_initialize`

- `model_step`
- "Launch the Model Interface Dialog Boxes"

Block parameter visibility

Specify whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class.

Settings

Default: `private`

`public`

> Generates the block parameter structure as a `public` data member of the C++ model class.

`private`

> Generates the block parameter structure as a `private` data member of the C++ model class.

`protected`

> Generates the block parameter structure as a `protected` data member of the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++` and **Code interface packaging** set to `C++ class`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `ParameterMemberVisibility`
Type: string
Value: `'public'` | `'private'` | `'protected'`
Default: `'private'`

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	`protected`

See Also

"Configure Code Interface Options"

Internal data visibility

Specify whether to generate internal data structures such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states as `public`, `private`, or `protected` data members of the C++ model class.

Settings

Default: `private`

`public`

Generates internal data structures as `public` data members of the C++ model class.

`private`

Generates internal data structures as `private` data members of the C++ model class.

`protected`

Generates internal data structures as `protected` data members of the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++` and **Code interface packaging** set to `C++ class`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `InternalMemberVisibility`
Type: string

Value: `'public'` | `'private'` | `'protected'`
Default: `'private'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	`protected`

See Also

"Configure Code Interface Options"

Block parameter access

Specify whether to generate access methods for block parameters for the C++ model class.

Settings

Default: None

None

   Does not generate access methods for block parameters for the C++ model class.

Method

   Generates noninlined access methods for block parameters for the C++ model class.

Inlined method

   Generates inlined access methods for block parameters for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++` and **Code interface packaging** set to `C++ class`.

- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `GenerateParameterAccessMethods`
Type: string
Value: `'None'` | `'Method'` | `'Inlined method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`
Efficiency	`Inlined method`
Safety precaution	`None`

See Also

"Configure Code Interface Options"

Internal data access

Specify whether to generate access methods for internal data structures, such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states, for the C++ model class.

Settings

Default: None

None

Does not generate access methods for internal data structures for the C++ model class.

Method

Generates noninlined access methods for internal data structures for the C++ model class.

Inlined method

Generates inlined access methods for internal data structures for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++` and **Code interface packaging** set to `C++ class`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `GenerateInternalMemberAccessMethods`
Type: string
Value: `'None'` | `'Method'` | `'Inlined method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`
Efficiency	`Inlined method`
Safety precaution	`None`

See Also

"Configure Code Interface Options"

External I/O access

Specify whether to generate access methods for root-level I/O signals for the C++ model class.

Note: This parameter affects generated code only if you are using the default (`void-void` style) step method for your model class; *not* if you are explicitly passing arguments for root-level I/O signals using an I/O arguments style step method. For more information, see "Passing No Arguments (void-void)" and "Passing I/O Arguments".

Settings

Default: `None`

`None`

> Does not generate access methods for root-level I/O signals for the C++ model class.

`Method`

> Generates noninlined access methods for root-level I/O signals for the C++ model class. The software generates set and get methods for each signal.

`Inlined method`

> Generates inlined access methods for root-level I/O signals for the C++ model class. The software generates set and get methods for each signal.

`Structure-based method`

> Generates noninlined, structure-based access methods for root-level I/O signals for the C++ model class. The software generates one set method, taking the address of the external input structure as an argument, and for external outputs (if applicable), one get method, returning the reference to the external output structure.

`Inlined structure-based method`

> Generates inlined, structure-based access methods for root-level I/O signals for the C++ model class. The software generates one set method, taking the address of the external input structure as an argument, and for external outputs (if applicable), one get method, returning the reference to the external output structure.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++` and **Code interface packaging** set to `C++ class`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: `GenerateExternalIOAccessMethods`
Type: string
Value: `'None'` | `'Method'` | `'Inlined method'` | `'Structure-based method'` | `'Inlined structure-based method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`

Application	Setting
Efficiency	Inlined method
Safety precaution	None

See Also

"Configure Code Interface Options"

Generate destructor

Specify whether to generate a destructor for the C++ model class.

Settings

Default: on

☑ On

   Generates a destructor for the C++ model class.

☐ Off

   Does not generate a destructor for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information
Parameter: GenerateDestructor
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	Off

See Also

"Configure Code Interface Options"

Configure C++ Class Interface

Open the Configure C++ class interface dialog box. In this dialog box, you can customize the C++ class interface for your model code. Based on your selections, you can preview and modify the model-specific C++ class interface.

Dependencies

- This button appears only for ERT-based targets with **Language** set to C++ and **Code interface packaging** set to C++ class.

- This button requires an Embedded Coder license when generating code.

- This button is active only if your model uses an attached configuration set. If your model uses a referenced configuration set, the button is greyed out. If you want to configure a model-specific C++ class interface for a referenced configuration set, use the MATLAB C++ class interface control functions described in "Customize C++ Class Interfaces Programmatically".

See Also

- "C++ Class Interface Control"
- model_step
- "Configure Step Method for Your Model Class"

MAT-file logging

Specify MAT-file logging

Settings

Default: on for the GRT target, off for ERT-based targets

☑ On

Enable MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in one of the following ways:

- **Configuration Parameters** > **Data Import/Export**, **Save to workspace** subpane (see "Data Import/Export Pane")
- To Workspace blocks
- To File blocks
- Scope blocks with the **Save data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in "Export Simulation Data" and "Configure Signal Data for Logging". Setting MAT-file logging redirects the data to a MAT-file instead. The file is named *model*.mat, where *model* is the name of your model.

☐ Off

Disable MAT-file logging. Clearing this option has the following benefits:

- Eliminates overhead associated with supporting a file system, which typically is not a requirement for embedded applications
- Eliminates extra code and memory usage required to initialize, update, and clean up logging variables
- Under certain conditions, eliminates code and storage associated with root output ports
- Omits the comparison between the current time and stop time in the *model*_step, allowing the generated program to run indefinitely, regardless of the stop time setting

Dependencies

- For the GRT target, selecting this option also selects the required option **Support non-finite numbers**. If you subsequently clear **Support non-finite numbers**, an error is displayed during code generation.
- For ERT-based targets, selecting this option also selects the required options **Support: floating-point numbers**, **Support: non-finite numbers**, and **Terminate function required**. If you subsequently clear **Support: floating-point numbers**, **Support: non-finite numbers**, or **Terminate function required**, an error is displayed during code generation.

4-163

- For ERT-based targets, selecting this option clears the incompatible option **Suppress error status in real-time model data structure**. If you subsequently select **Suppress error status in real-time model data structure**, an error is displayed during code generation.

- Selecting this option enables **MAT-file variable name modifier**.

- For ERT-based targets, clear this option if you are using exported function calls.

Limitations

In a referenced model, only the following data logging features are supported:

- To File blocks

- State logging — the software stores the data in the MAT-file for the top model.

In the context of the Embedded Coder product, MAT-file logging does not support the following IDEs: Analog Devices™ VisualDSP++®, Texas Instruments™ Code Composer Studio™, Wind River® DIAB/GCC.

Command-Line Information
Parameter: MatFileLogging
Type: string
Value: 'on' | 'off'
Default: 'on' for the GRT target, 'off' for ERT-based targets

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	Off
Safety precaution	Off

See Also

- "Logging"
- "Log Data for Analysis"
- "Virtualized Output Ports Optimization""Virtualized Output Ports Optimization"

MAT-file variable name modifier

Select the string to add to MAT-file variable names.

Settings

Default: rt_

rt_
>   Adds a prefix string.

_rt
>   Adds a suffix string.

none
>   Does not add a string.

Dependency

If you have an Embedded Coder license, for the GRT target or ERT-based targets, this parameter is enabled by **MAT-file logging**.

Command-Line Information
Parameter: LogVarNameModifier
Type: string
Value: 'none' | 'rt_' | '_rt'
Default: 'rt_'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Logging"
- "Log Data for Analysis"

4-165

Interface

Specify a data exchange interface to include in the generated code.

Settings

Default: None

None

    Does not generate extra code to support a data exchange interface.

C API

    Generates code for the C API data interface.

External mode

    Generates code for the External mode data interface.

ASAP2

    Generates code for the ASAP2 data interface.

Dependencies

Selecting **C API** enables the following parameters:

- **Generate C API for: signals**
- **Generate C API for: parameters**
- **Generate C API for: states**
- **Generate C API for: root-level I/O**

Selecting **External mode** enables the following parameters:

- **Transport layer**
- **MEX-file arguments**
- **Static memory allocation**

Command-Line Information
Parameter: see table
Type: string
Value: 'on' | 'off'
Default: 'off'

To enable...	Set this parameter...	To this value...
None	RTWCAPIParams, RTWCAPISignals, RTWCAPIStates, RTWCAPIRootIO, ExtMode, GenerateASAP2	'off'
C API	RTWCAPIParams, RTWCAPISignals, RTWCAPIStates, RTWCAPIRootIO	'on'
External mode	ExtMode	'on'
ASAP2	GenerateASAP2	'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development None for production code generation

See Also

- "Data Interchange Using the C API"
- "Host/Target Communication"
- "ASAP2 Data Measurement and Calibration"

Generate C API for: signals

Generate a C API signals structure.

Settings

Default: on

☑ On

Generates C API interface to global block outputs.

☐ Off

Does not generate C API signals.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information
Parameter: RTWCAPISignals
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Data Interchange Using the C API"

Generate C API for: parameters

Generate C API parameter tuning structures.

Settings

Default: on

☑ On

Generates C API interface to global block parameters.

☐ Off

    Does not generate C API parameters.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information
Parameter: `RTWCAPIParams`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Data Interchange Using the C API"

Generate C API for: states

Generate a C API states structure.

Settings

Default: off

☑ On

    Generates C API interface to discrete and continuous states.

☐ Off

    Does not generate C API states.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information
Parameter: RTWCAPIStates
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Data Interchange Using the C API"

Generate C API for: root-level I/O

Generate a C API root-level I/O structure.

Settings

Default: off

☑ On

Generates a C API interface to root-level inputs and outputs.

☐ Off

Does not generate a C API interface to root-level inputs and outputs.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information

Parameter: RTWCAPIRootIO
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Data Interchange Using the C API"

Transport layer

Specify the transport protocol for communications.

Settings

Default: tcpip

tcpip

    Applies a TCP/IP transport mechanism. The MEX-file name is ext_comm.

serial

    Applies a serial transport mechanism. The MEX-file name is
    ext_serial_win32_comm.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the
Configuration Parameters dialog box. The value is specified either in *matlabroot*/
toolbox/simulink/simulink/extmode_transports.m, for targets provided by
MathWorks®, or in an sl_customization.m file, for custom targets and/or custom
transports.

Dependency

This parameter is enabled by selecting `External mode` in the **Interface** parameter.

Command-Line Information
Parameter: `ExtModeTransport`
Type: integer
Value: `0` for TCP/IP | `1` for serial
Default: `0`

Recommended Settings

Application	No impact
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Target Interfacing"
- "Create a Transport Layer for External Communication"

MEX-file arguments

Specify arguments to pass to an External mode interface MEX-file for communicating with executing targets.

Settings

Default: `' '`

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target (for example, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

For a serial transport, `ext_serial_win32_comm` allows three optional arguments:

- Verbosity level (`0` for no information or `1` for detailed information)
- Serial port ID (for example, `1` for `COM1`, and so on)
- Baud rate (selected from the set `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600`, `115200`, with a default baud rate of `57600`)

Dependency

Depending on the specified "System target file", this parameter is enabled by the value selection **Data exchange** > **Interface** > **External mode** or by an **External mode** check box.

Command-Line Information
Parameter: `ExtModeMexArgs`
Type: string
Value: valid arguments
Default: `' '`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Target Interfacing"
- "Choose Communication Protocol for Client and Server"

Static memory allocation

Control memory buffer for External mode communication.

Settings

Default: off

☑ On

Enables the **Static memory buffer size** parameter for allocating dynamic memory.

☐ Off

Uses a static memory buffer for External mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- Depending on the specified "System target file", this parameter is enabled by the value selection **Data exchange** > **Interface** > **External mode** or by an **External mode** check box.

- This parameter enables **Static memory buffer size**.

Command-Line Information
Parameter: ExtModeStaticAlloc
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure External Mode Options for Code Generation"

Static memory buffer size

Specify the memory buffer size for External mode communication.

Settings

Default: 1000000

Enter the number of bytes to preallocate for External mode communications buffers in the target.

Tips

- If you enter too small a value for your application, External mode issues an out-of-memory error.
- To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information
Parameter: ExtModeStaticAllocSize
Type: integer
Value: valid value
Default: 1000000

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure External Mode Options for Code Generation"

4-175

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.

Parameter loading

☑ Enable RSim executable to load parameters from a MAT-file

Solver

Solver selection: | auto ▾ |

Storage classes

☑ Force storage classes to AUTO

In this section...
"Code Generation: RSim Target Tab Overview" on page 4-176
"Enable RSim executable to load parameters from a MAT-file" on page 4-177
"Solver selection" on page 4-177
"Force storage classes to AUTO" on page 4-178

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 4-4.

See Also

- "Configure and Build Model for Rapid Simulation"
- "Run Rapid Simulations"
- "Code Generation Pane: RSim Target" on page 4-176

Enable RSim executable to load parameters from a MAT-file

Specify whether to load RSim parameters from a MAT-file.

Settings

Default: on

☑ On

   Enables RSim to load parameters from a MAT-file.

☐ Off

   Disables RSim from loading parameters from a MAT-file.

Command-Line Information
Parameter: RSIM_PARAMETER_LOADING
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Create a MAT-File That Includes a Model Parameter Structure"

Solver selection

Instruct the target how to select the solver.

Settings

Default: auto

`auto`

> Lets the target choose the solver. The target uses the Simulink solver module if you specify a variable-step solver on the Solver pane. Otherwise, the target uses a Simulink Coder built-in solver.

`Use Simulink solver module`

> Instructs the target to use the variable-step solver that you specify on the **Solver** pane.

`Use fixed-step solvers`

> Instructs the target to use the fixed-step solver that you specify on the **Solver** pane.

Command-Line Information
Parameter: RSIM_SOLVER_SELECTION
Type: string
Value: `'auto'` | `'usesolvermodule'` | `'usefixstep'`
Default: `'auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Force storage classes to AUTO

Specify whether to retain your storage class settings in a model or to use the automatic settings.

Settings

Default: on

☑ On

> Forces the Simulink software to determine storage classes.

☐ Off

Causes the model to retain storage class settings.

Tips

- Turn this parameter on for flexible custom code interfacing.
- Turn this parameter off to retain storage class settings such as `ExportedGlobal` or `ImportExtern`.

Command-Line Information

Parameter: `RSIM_STORAGE_CLASS_AUTO`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Code Generation Pane: S-Function Target

The **Code Generation** > **S-Function Target** pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `rtwsfcn.tlc` system target file.

☑ Create new model

☐ Use value for tunable parameters

☐ Include custom source code

In this section...
"Code Generation S-Function Target Tab Overview" on page 4-180
"Create new model" on page 4-180
"Use value for tunable parameters" on page 4-181
"Include custom source code" on page 4-182

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 4-4.

See Also

- "Generated S-Function Block"
- "Code Generation Pane: S-Function Target" on page 4-180

Create new model

Create a new model containing the generated S-function block.

Settings

Default: on

☑ On

> Creates a new model, separate from the current model, containing the generated S-function block.

☐ Off

> Generates code but a new model is not created.

Command-Line Information
Parameter: `CreateModel`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

See Also

"Generated S-Function Block"

Use value for tunable parameters

Use the variable value instead of the variable name in generated block mask edit fields for tunable parameters.

Settings

Default: off

☑ On

> Uses variable values for tunable parameters instead of the variable name in the generated block mask edit fields.

☐ Off

> Uses variable names for tunable parameters in the generated block mask edit fields.

Command-Line Information
Parameter: `UseParamValues`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Generated S-Function Block"

Include custom source code

Include custom source code in the code generated for the S-function.

Settings

Default: off

☑ On

Include provided custom source code in the code generated for the S-function.

☐ Off

Do not include custom source code in the code generated for the S-function.

Command-Line Information
Parameter: `AlwaysIncludeCustomSrc`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Generated S-Function Block"

Code Generation Pane: Tornado Target

The **Code Generation** > **Tornado Target** pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `tornado.tlc` system target file.

In this section...

In this section...

Code Generation: Tornado Target Tab Overview

Control Simulink Coder generated code for the Tornado target.

Configuration

This tab appears only if you specify `tornado.tlc` as the "System target file" on page 4-4.

See Also

- *Tornado User's Guide* from Wind River Systems
- *StethoScope User's Guide* from Wind River Systems
- "Asynchronous Support"
- "Code Generation Pane: Tornado Target" on page 4-183

Standard math library

Specify a standard math library for your model.

Settings

Default: `C89/C90 (ANSI)`

`C89/C90 (ANSI)`

　　Generates calls to the ISO/IEC 9899:1990 C standard math library for floating-point functions.

`C99 (ISO)`

　　Generates calls to the ISO/IEC 9899:1999 C standard math library.

```
C++03 (ISO)
```
> Generates calls to the ISO/IEC 14882:2003 C++ standard math library.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Dependencies

The `C++03 (ISO)` math library is available for use only if you select `C++` for the **Language** parameter.

Command-Line Information
Parameter: `TargetLangStandard`
Type: string
Value: `'C89/C90 (ANSI)'` | `'C99 (ISO)'` | `'C++03 (ISO)'`
Default: `'C89/C90 (ANSI)'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Specify Target Interfaces"

Code replacement library

Specify an application-specific math library for your model.

Settings

Default: None

None

Does not use a code replacement library.

`GNU C99 extensions`

Generates calls to the GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

`Intel IPP for x86-64 (Windows)`

Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform.

`Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)`

Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Windows platform.

`Intel IPP for x86/Pentium (Windows)`

Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.

`Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)`

Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform.

`Intel IPP for x86-64 (Linux)`

Generates calls to the Intel Performance Primitives (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 Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Linux platform.

Note:

- Additional values might be listed for licensed target products, for embedded and desktop targets, or if you have created and registered code replacement libraries using the Embedded Coder product.

- The list of **Code replacement library** values is filtered based on compatibility with the **Language**, **Standard math library**, and **Device vendor** values selected for your model.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Command-Line Information

Parameter: CodeReplacementLibrary
Type: string
Value: 'None' | 'GNU C99 extensions' | 'Intel IPP' | 'Intel IPP/SSE with GNU99 extensions'
Default: 'None'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Valid library
Safety precaution	No impact

See Also

"Specify Target Interfaces"

Shared code placement

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Settings

Default: Auto

Auto

Operates as follows:

- When the model contains Model blocks, places utility code within the slprj/
  target/_sharedutils folder.
- When the model does not contain Model blocks, places utility code in the build
  folder (generally, in *model*.c or *model*.cpp).

```
Shared location
```

    Directs code for utilities to be placed within the `slprj` folder in your working folder.

Command-Line Information
Parameter: `UtilityFuncGeneration`
Type: string
Value: `'Auto'` | `'Shared location'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	`Shared location`
Traceability	`Shared location`
Efficiency	No impact (execution, RAM) `Shared location` (ROM)
Safety precaution	No impact

See Also

- "Specify Target Interfaces"
- "Sharing Utility Code"

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: off

☑ On

    Enables MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in one of the following ways:

- Configuration Parameters dialog box, **Data Import/Export** pane, **Save to workspace** subpane (see "Data Import/Export Pane")
- To Workspace blocks

- Scope blocks with the **Save data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in "Export Simulation Data" and "Configure Signal Data for Logging". Setting MAT-file logging redirects the data to a MAT-file instead. The file is named *model*.mat, where *model* is the name of your model.

☐ Off

Disables MAT-file logging. Clearing this option has the following benefits:

- Eliminates overhead associated with supporting a file system, which typically is not required for embedded applications
- Eliminates extra code and memory usage required to initialize, update, and clean up logging variables
- Under certain conditions, eliminates code and storage associated with root output ports
- Omits the comparison between the current time and stop time in the *model*_step, allowing the generated program to run indefinitely, regardless of the stop time setting

Dependencies

Selecting this parameter enables **MAT-file variable name modifier**.

Limitation

MAT-file logging does not work in a referenced model, and code is not generated to implement it.

Command-Line Information
Parameter: MatFileLogging
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact

Application	Setting
Efficiency	Off
Safety precaution	Off

See Also

- "Logging"
- "Log Data for Analysis"
- "Virtualized Output Ports Optimization"

MAT-file variable name modifier

Select the string to add to the MAT-file variable names.

Settings

Default: rt_

rt_
   Adds a prefix string.

_rt
   Adds a suffix string.

none
   Does not add a string.

Dependency

If you have an Embedded Coder license, this parameter is enabled by **MAT-file logging**.

Command-Line Information
Parameter: LogVarNameModifier
Type: string
Value: 'none' | 'rt_' | '_rt'
Default: 'rt_'

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Logging"
- "Log Data for Analysis"

Code Format

Specify the code generation format.

Settings

Default: RealTime

RealTime

Specifies the Real-Time code generation format.

RealTimeMalloc

Specifies the Real-Time Malloc code generation format.

Command-Line Information

Parameter: CodeFormat
Type: string
Value: 'RealTime' | 'RealTimeMalloc'
Default: 'RealTime'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

4-191

See Also

"Targets and Code Formats"

StethoScope

Specify whether to enable StethoScope, an optional data acquisition and data monitoring tool.

Settings

Default: off

☑ On

   Enables StethoScope.

☐ Off

   Disables StethoScope.

Tips

You can optionally monitor and change the parameters of the executing real-time program using either StethoScope or Simulink External mode, but not both with the same compiled image.

Dependencies

Enabling **StethoScope** automatically disables **External mode**, and vice versa.

Command-Line Information
Parameter: StethoScope
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	On

Application	Setting
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: string
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 Simulink Coder software increments the priority of each subrate task by one.
- The value you specify for this option will be overridden by a base priority specified in a call to the rt_main() function spawned as a task.

Command-Line Information
Parameter: BasePriority
Type: integer
Value: valid value
Default: 30

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	Might impact efficiency, depending on other task's priorities
Safety precaution	No impact

See Also

- *Tornado User's Guide* from Wind River Systems
- "Asynchronous Support"

Task stack size

Stack size in bytes for each task that executes the model.

Settings

Default: 16384

Command-Line Information
Parameter: `TaskStackSize`
Type: integer
Value: valid value
Default: `16384`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Larger stack may waste space
Safety precaution	Larger stack reduces the possibility of overflow

See Also

- *Tornado User's Guide* from Wind River Systems
- "Asynchronous Support"

External mode

Specify whether to enable communication between the Simulink model and an application based on a client/server architecture.

Settings

Default: on

☑ On

> Enables External mode. The client (Simulink model) transmits messages requesting the server (application) to accept parameter changes or to upload signal data. The server responds by executing the request.

☐ Off

> Disables External mode.

Dependencies

Selecting this parameter enables:

- **Transport layer**
- **MEX-file arguments**
- **Static memory allocation**

Command-Line Information
Parameter: `ExtMode`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Host/Target Communication"

Transport layer

Specify the transport protocol for External mode communications.

Settings

Default: `tcpip`

`tcpip`

    Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. For targets provided by MathWorks, the value is specified in *matlabroot*/toolbox/simulink/simulink/extmode_transports.m.

Dependency

This parameter is enabled by the **External mode** check box.

Command-Line Information
Parameter: `ExtModeTransport`
Type: integer
Value: `0`
Default: `0`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Target Interfacing"

MEX-file arguments

Specify arguments to pass to an External mode interface MEX-file for communicating with executing targets.

Settings

Default: `''`

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- Network name of your target (for example, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

Dependency

This parameter is enabled by the **External mode** check box.

Command-Line Information
Parameter: `ExtModeMexArgs`
Type: string
Value: valid arguments
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Target Interfacing"
- "Choose Communication Protocol for Client and Server"

Static memory allocation

Control the memory buffer for External mode communication.

Settings

Default: off

☑ On

> Enables the **Static memory buffer size** parameter for allocating allocate dynamic memory.

☐ Off

> Uses a static memory buffer for External mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- This parameter is enabled by the **External mode** check box.
- This parameter enables **Static memory buffer size**.

Command-Line Information

Parameter: ExtModeStaticAlloc
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure External Mode Options for Code Generation"

Static memory buffer size

Specify the memory buffer size for External mode communication.

Settings

Default: 1000000

Enter the number of bytes to preallocate for External mode communications buffers in the target.

Tips

- If you enter too small a value for your application, External mode issues an out-of-memory error.
- To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information
Parameter: ExtModeStaticAllocSize
Type: integer
Value: valid value
Default: 1000000

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Configure External Mode Options for Code Generation"

Code Generation: Coder Target Pane

In this section...

In this section...

Code Generation: Coder Target Pane Overview (previously "IDE Link Tab Overview")

Configure the parameters for:

- Tool Chain Automation — How the coder software interacts with third-party software build toolchains.
- Target Hardware Resources — The IDE toolchain and properties of the physical hardware, such as board, operating system, memory, and peripherals.

See Also

- Coder Target: Tool Chain Automation Tab Overview
- Coder Target: Target Hardware Resources Tab Overview

Coder Target: Tool Chain Automation Tab Overview
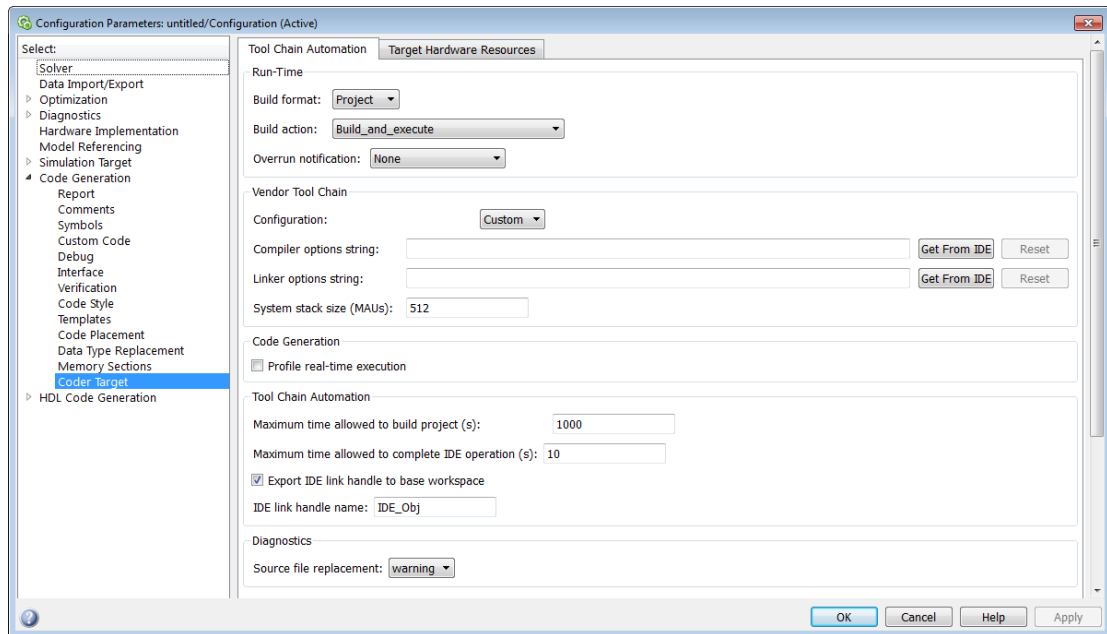
The Tool Chain Automation Tab is only visible under the Coder Target pane.

The following table lists the parameters on the Tool Chain Automation Tab.

Build format

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: `Project`

`Project`

    Builds your model as an IDE project.

`Makefile`

    Creates a makefile and uses it to build your model.

Dependencies

Selecting `Makefile` removes the following parameters:

- **Code Generation**

  - **Profile real-time execution**
  - **Profile by**
  - **Number of profiling samples to collect**

- **Link Automation**

  - **Maximum time allowed to build project (s)**
  - **Maximum time allowed to complete IDE operation (s)**
  - **Export IDE link handle to base workspace**
  - **IDE link handle name**

Command-Line Information
Parameter: `buildFormat`

Type: string
Value: `Project | Makefile`
Default: `Build_and_execute`

Recommended Settings

Application	Setting
Debugging	`Project`
Traceability	`Project`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: `Build_and_execute`

If you set **Build format** to `Project`, select one of the following options:

`Build_and_execute`

Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

`Create_project`

Directs Simulink Coder software to create a new project in the IDE. The command line equivalent for this setting is `Create`.

`Archive_library`

Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

`Build`

Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

Create_processor_in_the_loop_project

Directs the Simulink Coder code generation process to create PIL algorithm object code as part of the project build.

If you set **Build format** to Makefile, select one of the following options:

Create_makefile

Creates a makefile. For example, ".mk". The command line equivalent for this setting is Create.

Archive_library

Creates a makefile and an archive library. For example, ".a" or ".lib".

Build

Creates a makefile and an executable. For example, ".exe".

Build_and_execute

Creates a makefile and an executable. Then it evaluates the execute instruction under the **Execute** tab in the current XMakefile configuration.

Dependencies

Selecting Archive_library removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace**

Selecting Create_processor_in_the_loop_project removes the following parameters:

- **Overrun notification**

- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

Command-Line Information

Parameter: `buildAction`
Type: string
Value: `Build` | `Build_and_execute` | `Create` | `Archive_library` | `Create_processor_in_the_loop_project`
Default: `Build_and_execute`

Recommended Settings

Application	Setting
Debugging	`Build_and_execute`
Traceability	`Archive_library`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

For more information about PIL and its uses, refer to the "Verifying Generated Code via Processor-in-the-Loop" topic.

Overrun notification

Specifies how your program responds to overrun conditions during execution.

Settings

Default: `None`

`None`

> Your program does not notify you when it encounters an overrun condition.

`Print_message`

> Your program prints a message to standard output when it encounters an overrun condition.

`Call_custom_function`

> When your program encounters an overrun condition, it executes a function that you specify in **Function name**.

Tips

- The definition of the standard output depends on your configuration.

Dependencies

Selecting `Call_custom_function` enables the **Function name** parameter.

Setting this parameter to `Call_custom_function` enables the **Function name** parameter.

Command-Line Information
Parameter: `overrunNotificationMethod`
Type: string
Value: `None` | `Print_message` | `Call_custom_function`
Default: `None`

Recommended Settings

Application	Setting
Debugging	`Print_message` or `Call_custom_function`
Traceability	`Print_message`
Efficiency	`None`
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Function name

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

Settings

No Default

Dependencies

This parameter is enabled by setting **Overrun notification** to `Call_custom_function`.

Command-Line Information
Parameter: `overrunNotificationFcn`
Type: string
Value: no default
Default: no default

Recommended Settings

Application	Setting
Debugging	String
Traceability	String
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Configuration

Sets the Configuration for building your project from the model.

Settings

Default: `Custom`

Custom

> Lets the user apply a specialized combination of build and optimization settings.
>
> Custom applies the same settings as the Release project configuration in IDE, except:
>
> - The compiler options do not use optimizations.
> - The memory configuration specifies a memory model that uses Far Aggregate for data and Far for functions.

Debug

> Applies the Debug Configuration defined by the IDE to the generated project and code.

Release

> Applies the Release project configuration defined by the IDE to the generated project and code.

Dependencies

- Selecting Custom disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting Release sets the **Compiler options string** to the settings defined by the IDE.
- Selecting Debug sets the **Compiler options string** to the settings defined by the IDE.

.

Command-Line Information
Parameter: projectOptions
Type: string
Value: Custom | Debug | Release
Default: Custom

Recommended Settings

Application	Setting
Debugging	Custom or Debug
Traceability	Custom, Debug, Release

Application	Setting
Efficiency	`Release`
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the coder product does not set optimization flags.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Configuration** to `Custom` applies the `Custom` compiler options defined by coder software. `Custom` does not use optimizations.
- Setting **Configuration** to `Debug` applies the debug settings defined by the IDE.
- Setting **Configuration** to `Release` applies the release settings defined by the IDE.

Command-Line Information
Parameter: `compilerOptionsStr`
Type: string
Value: `Custom` | `Debug` | `Release`
Default: `Custom`

Recommended Settings

Application	Setting
Debugging	`Custom`

Application	Setting
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 coder product does not set linker options.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information
Parameter: `linkerOptionsStr`
Type: string
Value: valid linker option
Default: none

Recommended Settings

Application	Setting
Debugging	No impact

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.

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 coder software must create a handle.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop project` removes this parameter.

Command-Line Information
Parameter: `ProfileGenCode`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

For more information about using profiling, refer to the "profile" and "Profiling Code Execution in Real-Time" topics..

Profile by

Defines which execution profiling technique to use.

Settings

Default: `Task`

`Task`

> Profiles model execution by the tasks in the model.

`Atomic subsystem`

> Profiles model execution by the atomic subsystems in the model.

Dependencies

Selecting **Real-time execution profiling** enables this parameter.

Command-Line Information
Parameter: `profileBy`
Type: string
Value: `Task` | `Atomic subsystem`
Default: `Task`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

For more information about PIL and its uses, refer to the "Verifying Generated Code via Processor-in-the-Loop" topic.

For more information about using profiling, refer to the "profile" and "Profiling Code Execution in Real-Time" topics.

Number of profiling samples to collect

Specify the size of the buffer that holds the profiling samples. Enter a value that is 2 times the number of profiling samples.

Each task or subsystem execution instance represents one profiling sample. Each sample requires two memory locations, one for the start time and one for the end time. Consequently, the size of the buffer is twice the number of samples.

Sample collection begins with the start of code execution and ends when the buffer is full.

The profiling data is held in a statically sited buffer on the target processor.

Settings

Default: 100

Minimum: 2

Maximum: Buffer capacity

Tips

- Data collection stops when the buffer is full, but the application and processor continue running.
- Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

Dependencies

This parameter is enabled by **Profile real-time execution**.

Command-Line Information
Parameter:`ProfileNumSamples`
Type: int
Value: Positive integer
Default: 100

Recommended Settings

Application	Setting
Debugging	100
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

Settings

Default: 1000

Minimum: 1

Maximum: No limit

Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to complete IDE operation** timeout value.

Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

Command-Line Information
Parameter:`ideObjBuildTimeout`
Type: int
Value: Integer greater than 0
Default: 100

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Maximum time allowed to complete IDE operation (s)

specifies how long, in seconds, the software waits for IDE functions, such as `read` or `write`, to return completion messages.

Settings

Default: 10

Minimum: 1

Maximum: No limit

Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to build project (s)** timeout value

Command-Line Information
Parameter: `'ideObjTimeout'`
Type: int
Value:
Default: 10

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Export IDE link handle to base workspace

Directs the software to export the `IDE_Obj` object to your MATLAB workspace.

Settings

Default: On

☑ On

> Directs the build process to export the `IDE_Obj` object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.

☐ Off

> prevents the build process from exporting the `IDE_Obj` object to your MATLAB software workspace.

Dependency

Selecting **Profile real-time execution** enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Selecting **Export IDE link handle to base workspace** enables **IDE link handle name**.

Command-Line Information
Parameter: `exportIDEObj`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

IDE link handle name

specifies the name of the `IDE_Obj` object that the build process creates.

Settings

Default: `IDE_Obj`

- Enter a valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the `IDE_Obj` object.
- The handle name is case sensitive.

Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

Command-Line Information
Parameter: `ideObjName`
Type: string
Value:
Default: `IDE_Obj`

Recommended Settings

Application	Setting
Debugging	Enter a valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact

Application	Setting
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Source file replacement

Selects the diagnostic action to take if the coder software detects conflicts that you are replacing source code with custom code.

Settings

Default: warn

> Does not generate warnings or errors when it finds conflicts.

warning

> Displays a warning.

error

> Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

Tips

- The build operation continues if you select warning and the software detects custom code replacement. You see warning messages as the build progresses.
- Select error the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select none when you do not want to see multiple messages during your build.
- The messages apply to Simulink Coder **Custom Code** replacement options as well.

Command-Line Information
Parameter: DiagnosticActions
Type: string
Value: none | warning | error

Default: `warning`

Recommended Settings

Application	Setting
Debugging	`error`
Traceability	`error`
Efficiency	`warning`
Safety precaution	`error`

See Also

For more information, refer to the "Code Generation Pane: Coder Target" topic.

Parameter Reference

In this section...
"Recommended Settings Summary" on page 4-224
"Parameter Command-Line Information Summary" on page 4-248

Recommended Settings Summary

The following table summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the GRT and ERT targets, unless otherwise specified.

For parameters that are available only when an ERT target is specified, see the "Recommended Settings Summary" in the Embedded Coder documentation.

For additional details, click the links in the Configuration Parameter column.

Mapping Application Requirements to the Solver Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"**Start time**"	No impact	No impact	No impact	0.0	0.0 seconds
"**Stop time**"	No impact	No impact	No impact	A positive value	10.0 seconds
"**Type**"	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Fixed-step`	`Variable-step` (you must change to `Fixed-step` for code generation)
"**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	Unconstrained

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
				time independent	
"Sample time properties"	No impact	No impact	No impact	Period, offset, and priority of each sample time in the model; faster sample times must have higher priority than slower sample times	''
"Tasking mode for periodic sample times"	No impact	No impact	No impact	No impact	Auto
"Automatically handle rate transition for data transfer"	No impact	No impact (for simulation and during development) Off (for production code generation)	No impact	Off	Off

Mapping Application Requirements to the Data Import/Export Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Input"	No impact	No impact	No impact	No impact (GRT)	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
				Off (ERT)	
"Initial state"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
"Time"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On
"States"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
"Output"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On
"Final states"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
"Signal logging"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On
"Record logged workspace data in Simulation Data Inspector"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Off
"Limit data points to last"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Decimation"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	1
"Format"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Array
"Output options"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	Refine output
"Refine factor"	No impact	No impact	No impact	No impact (GRT) Off (ERT)	1
"Output times "	No impact	No impact	No impact	No impact (GRT) Off (ERT)	'[]'

Mapping Application Requirements to the Optimization Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Block reduction"	Off (GRT) No impact (ERT)	Off	On	Off	On
"Implement logic signals as Boolean data (vs. double)"	No impact	No impact	On	On	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	**Traceability**	**Efficiency**	**Safety precaution**	
"**Conditional input branch execution**"	No impact	On	On (execution) No impact (ROM, RAM)	No impact	On
"**Application lifespan (days)**"	No impact	No impact	Finite value	`inf`	`inf`
"**Use memset to initialize floats and doubles to 0.0**"	No impact	No impact	On* (execution, ROM) No impact (RAM)	No impact	On
"**Use floating-point multiplication to handle net slope corrections**"	No impact	No impact	On (when target hardware supports efficient multiplicati Off (otherwise)	Off	Off
"**Remove code from floating-point to integer conversions that wraps out-of-range values**"	Off	Off	On (execution, ROM) No impact (RAM)	Off (GRT) On (ERT)	Off
"**Remove code from floating-point to integer conversions with saturation that**"	Off	Off	On	Off (GRT) On (ERT)	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
maps NaN to zero"					
*The command-line value is reverse of the listed value.					

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Inline parameters"	Off (GRT) On (ERT)	On	On	No impact	Off
"Signal storage reuse"	Off	Off	On	No impact	On
"Enable local block outputs"	Off	No impact	On	No impact	On
"Eliminate superfluous local variables (Expression folding)"	Off	No impact (GRT) Off (ERT)	On	No impact	On
"Optimize global data access"	Off	Off	No impact (execution) On (ROM, RAM)	No impact	Off
"Loop unrolling threshold"	No impact	No impact	>0	>1	5
"Maximum stack size (bytes)"	No impact	No impact	No impact	No impact	Inherit from target
"Use memcpy for vector assignment"	No impact	No impact	On	No impact	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"**Memcpy threshold (bytes)**"	No impact	No impact	Accept default or determine target-specific optimal value	No impact	64
"**Reuse local block outputs**"	Off	Off	On	No impact	On
"**Inline invariant signals**"	Off	Off	On	No impact	Off

Mapping Application Requirements to the Optimization Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"**Use bitsets for storing state configuration**"	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off
"**Use bitsets for storing Boolean data**"	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off

Mapping Application Requirements to the Diagnostics Pane: Solver Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"**Algebraic loop**"	error	No impact	No impact	error	warning

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Minimize algebraic loop"	No impact	No impact	No impact	error	warning
"Block priority violation"	No impact	No impact	No impact	error	warning
"Consecutive zero-crossings violation"	No impact	No impact	No impact	warning or error	error
"Unspecified inheritability of sample time"	No impact	No impact	No impact	error	warning
"Solver data inconsistency"	warning	No impact	none	No impact	warning
"Automatic solver parameter selection"	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Sample Time Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Source block specifies -1 sample time"	No impact	No impact	No impact	error	none
"Discrete used as continuous"	No impact	No impact	No impact	error	warning
"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	No impact	No impact	No impact	error	error

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
executed subsystem"					
"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
"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

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Detect downcast"	No impact	No impact	No impact	error	error
"Detect overflow"	No impact	No impact	No impact	error	error
"Detect underflow"	No impact	No impact	No impact	error	none
"Detect precision loss"	No impact	No impact	No impact	error	error
"Detect loss of tunability"	No impact	No impact	No impact	error	none
"Detect read before write"	No impact	No impact	No impact	error	Enable all as warnings
"Detect write after read"	No impact	No impact	No impact	error	Enable all as warning
"Detect write after write"	No impact	No impact	No impact	error	Enable all as errors
"Multitask data store"	No impact	No impact	No impact	error	warning
"Duplicate data store names"	warning	No impact	none	No impact	none
"Check undefined subsystem initial output"	No impact	No impact	No impact	On	On
"Check preactivation output of execution context"	No impact	No impact	No impact	On	Off
"Check runtime output of	No impact	No impact	No impact	On	Off

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
execution context"					
"Model Verification block enabling"	No impact	No impact	No impact	No impact (GRT) Disable all (ERT)	Use local settings

Mapping Application Requirements to the Diagnostics Pane: Type Conversion Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Unnecessary type conversions"	No impact	No impact	No impact	warning	none
"Vector/matrix block input conversion"	No impact	No impact	No impact	error	none
"32-bit integer to single precision float conversion"	No impact	No impact	No impact	warning	warning

Mapping Application Requirements to the Diagnostics Pane: Connectivity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Signal label mismatch"	No impact	No impact	No impact	error	none
"Unconnected block input ports"	No impact	No impact	No impact	error	warning

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	none	none
"Port and parameter mismatch"	No impact	No impact	No impact	error	none
"Model configuration mismatch"	No impact	No impact	No impact	warning	none
"Invalid root Inport/ Outport block connection"	No impact	No impact	No impact	error	none
"Unsupported data logging"	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Saving Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Block diagram contains disabled library links"	No impact	No impact	No impact	No impact	warning
"Block diagram contains parameterized library links"	No impact	No impact	No impact	No impact	none

Mapping Application Requirements to the Diagnostics Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Unused data and events"	warning	No impact	No impact (for simulation and during development) none (for production code generation)	warning	warning
"Unexpected backtracking"	warning	No impact	No impact	error	warning
"Invalid input data access in chart initialization"	warning	No impact	No impact	error	warning
"No unconditional default transitions"	warning	No impact	No impact (for simulation and during development) none (for production code generation)	error	warning
"Transition outside natural parent"	warning	No impact	No impact (for simulation and during development) none (for production	error	warning

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
			code generation)		

Mapping Application Requirements to the Hardware Implementation Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"**Device vendor**"	No impact	No impact	No impact	No impact	`Generic`
"**Device type**"	No impact	No impact	No impact	No impact	`Unspecified (assume 32—bit Generic)`
"**Number of bits**"	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for code generation	`char` 8, `short` 16, `int` 32, `long` 32, `long long` 64, `float` 32, `double` 64, `native` 32, `pointer` 32
"**Largest atomic size**"	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for code generation	integer `Char`, floating-point `None`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"**Byte ordering**"	No impact	No impact	No impact	No impact	Unspecified
"**Signed integer division rounds to**"	No impact for simulation and during development Undefined for production code generation	No impact for simulation and during development Zero or Floor for production code generation	No impact for simulation and during development Zero for production code generation	No impact for simulation and during development Floor for production code generation	Undefined
"**Shift right on a signed integer as arithmetic shift**"	No impact	No impact	On	No impact	On
"**Enable long long**"	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for production code generation	Off
"**Test hardware is the same as production hardware**"	No impact	No impact	No impact	No impact	On

Mapping Application Requirements to the Model Referencing Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Rebuild"	No impact	No impact	No impact	`If any changes detected` or `Never` If you use the `Never` setting, then set the **Never rebuild diagnostic** parameter to `Error if rebuild required`	`If any changes detected`
"Never rebuild diagnostic"	No impact	No impact	No impact	`error if rebuild required`	`error if rebuild required`
"Enable parallel model reference builds"	No impact	No impact	No impact	No impact	`Off`
"MATLAB worker initialization for builds"	No impact	No impact	No impact	No impact	`None`
"Total number of instances allowed per top model"	No impact	No impact	No impact	No impact	`Multiple`
"Pass fixed-size scalar root inputs by value for code generation"	No impact	No impact	No impact	`Off`	`Off`

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Minimize algebraic loop occurrences"	No impact	No impact	No impact	Off	Off
"Propagate sizes of variable-size signals"	No impact	No impact	No impact	Off	`Infer from blocks in model`
"Model dependencies"	No impact	No impact	No impact	No impact	`''`

Mapping Application Requirements to the Simulation Target Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Enable debugging/ animation"	On	No impact	Off	On	On
"Detect wrap on overflow (with debugging)"	On	No impact	Off	On	On
"Ensure memory integrity"	On	On	Off	On	On
"Echo expressions without semicolons"	On	No impact	Off	No impact	On
"Ensure responsiveness"	On	On	Off	On	On
"Simulation target build mode"	No impact	No impact	No impact	No impact	`Incremental build`

Mapping Application Requirements to the Simulation Target Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	**Traceability**	**Efficiency**	**Safety precaution**	
"Reserved names"	No impact	No impact	No impact	No impact	{}

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	**Traceability**	**Efficiency**	**Safety precaution**	
"Parse custom code symbols"	On	No impact	No impact	On	On
"Source file"	No impact	No impact	No impact	No impact	' '
"Header file"	No impact	No impact	No impact	No impact	' '
"Initialize function"	No impact	No impact	No impact	No impact	' '
"Terminate function"	No impact	No impact	No impact	No impact	' '
"Include directories"	No impact	No impact	No impact	No impact	' '
"Source files"	No impact	No impact	No impact	No impact	' '
"Libraries"	No impact	No impact	No impact	No impact	' '

Mapping Application Requirements to the Code Generation Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	**Traceability**	**Efficiency**	**Safety precaution**	
System target file	No impact	No impact	No impact	No impact (GRT) ERT based (ERT)	grt.tlc
Language	No impact	No impact	No impact	No impact	C

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	**Traceability**	**Efficiency**	**Safety precaution**	
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
Make command	No impact	No impact	No impact	`make_rtw`	`make_rtw`
Template makefile	No impact	No impact	No impact	No impact	`grt_default_tmf`
"Select objective" on page 4-21	`Debugging`	Not applicable for GRT-based targets	`Execution efficiency`	Not applicable for GRT-based targets	`Unspecified`
"Check model before generating code" on page 4-26	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`Off`
Generate code only	`Off`	No impact	No impact	No impact	`Off`

Mapping Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
"Create code generation report" on page 4-33	On	On	No impact	On	Off
"Open report automatically" on page 4-36	On	On	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Comments Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Include comments	On	On	No impact	On	On
Simulink block / Stateflow object comments	On	On	No impact	On	On
Show eliminated blocks	On	On	No impact	On	Off
Verbose comments for Simulink Global storage class	On	On	No impact	On	Off
Operator Annotations	No impact	On	No impact	On	Off

Mapping Application Requirements to the Code Generation Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Maximum identifier length	Valid value	>30	No impact	>30	31
Use the same reserved names as Simulation Target	No impact	No impact	No impact	No impact	Off
Reserved names	No impact	No impact	No impact	No impact	{}

Mapping Application Requirements to the Code Generation Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Use the same custom code settings as Simulation Target	No impact	No impact	No impact	No impact	Off
Source file	No impact	No impact	No impact	No impact	' '
Header file	No impact	No impact	No impact	No impact	' '
Initialize function	No impact	No impact	No impact	No impact	' '
Terminate function	No impact	No impact	No impact	No impact	' '
Include directories	No impact	No impact	No impact	No impact	' '
Source files	No impact	No impact	No impact	No impact	' '
Libraries	No impact	No impact	No impact	No impact	' '

Mapping Application Requirements to the Code Generation Pane: Debug Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Verbose build	On	No impact	No impact	On	On
Retain .rtw file	On	No impact	No impact	No impact	Off
"Profile TLC" on page 4-112	On	No impact	No impact	No impact	Off
Start TLC debugger when generating code	On	No impact	No impact	No impact	Off
Start TLC coverage when generating code	On	No impact	No impact	No impact	Off
Enable TLC assertion	On	No impact	No impact	On	Off

Mapping Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Standard math library	No impact	No impact	Valid library	No impact	`C89/C90 (ANSI)`
Code replacement library	No impact	No impact	Valid library	No impact	`None`
Shared code placement	`Shared location` (GRT) No impact (ERT)	`Shared location` (GRT) No impact (ERT)	No impact (execution, RAM) `Shared location` (ROM)	No impact	`Auto`
Support non-finite numbers	No impact	No impact	Off (Execution, ROM)	Off	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	**Traceability**	**Efficiency**	**Safety precaution**	
			No impact (RAM)		
"**Code interface packaging**"	No impact	No impact	`Reusable function` or `C++ class`	No impact	`Nonreusable function` if **Language** is set to `C`; `C++ class` if **Language** is set to `C++`
"**Multi-instance code error diagnostic**"	`Warning` or `Error`	No impact	`None`	No impact	`Error`
"**Classic call interface**"	No impact	Off	Off (execution, ROM), No impact (RAM)	Off	Off (except On for GRT models created before R2012a)
Single output/ update function	On	On	On	On	On
MAT-file logging	On	No impact	Off	Off	On (GRT) \n\n Off (ERT)
MAT-file variable name modifier	No impact	No impact	No impact	No impact	`rt_`
Interface	No impact	No impact	No impact	No impact (GRT) \n\n `None` (ERT)	`None`
Generate C API for: signals	No impact	No impact	No impact	No impact	On
Generate C API for: parameters	No impact	No impact	No impact	No impact	On

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety precaution	
Generate C API for: states	No impact	No impact	No impact	No impact	Off
Generate C API for: root-level I/O	No impact	No impact	No impact	No impact	Off
Transport layer	No impact	No impact	No impact	No impact	`tcpip`
MEX-file arguments	No impact	No impact	No impact	No impact	`''`
Static memory allocation	No impact	No impact	No impact	No impact	Off
"Static memory buffer size" on page 4-200	No impact	No impact	No impact	No impact	1000000

Parameter Command-Line Information Summary

The following table lists Simulink Coder parameters that you can use to tune model and target configurations. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to Configuration Parameter dialog box equivalents.

Use the `get_param` and `set_param` commands to retrieve and set the values of the parameters on the MATLAB command line or programmatically in scripts.

The Configuration Wizard in the Embedded Coder product provides buttons and scripts for customizing code generation. For information on using Configuration Wizard features, see "Use Configuration Wizard Blocks" in the Embedded Coder documentation.

For general information about Simulink parameters, see "Configuration Parameters Dialog Box Overview". For information on using `get_param` and `set_param` to tune the parameters for various model configurations, see "Tune Parameters".

For parameters that are specific to the ERT target, or targets based on the ERT target, see "Parameter Command-Line Information Summary" in the Embedded Coder documentation.

> **Note** Parameters that are specific to Stateflow or Fixed-Point Designer™ products are marked with (Stateflow) and (Fixed-Point Designer), respectively.
>
> The default setting for a parameter might vary for different targets.

Command-Line Information: Optimization Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
BooleanDataType off, **on**	**Optimization > Implement logic signals as Boolean data (vs. double)**	Control the output data type of blocks that generate logic signals.
EfficientFloat2IntCast **off**, on	**Optimization > Remove code from floating-point to integer conversions that wrap out-of-range values**	Remove wrapping code that handles out-of-range floating-point to integer conversion results.
EfficientMapNaN2IntZero off, **on**	**Optimization > Remove code from floating-point to integer conversions with saturation that maps NaN to zero**	Remove code that handles floating-point to integer conversion results for NaN values.
InitFltsAndDblsToZero **off**, on	**Optimization > Use memset to initialize floats and doubles to 0.0**	Optimize initialization of storage for float and double values. Set this option if the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern 0.
LifeSpan *string*	**Optimization > Application lifespan (days)**	Optimize the size of counters used to compute absolute and elapsed time, using the specified application life span value.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`NoFixptDivByZeroProtection` (Fixed-Point Designer) **off**, on	**Optimization** > **Remove code that protects against division arithmetic exceptions**	Suppress generation of code that guards against division by zero for fixed-point data.
`UseFloatMulNetSlope` (Fixed-Point Designer) **off**, on	**Optimization** > **Use floating-point multiplication to handle net slope corrections**	Use floating-point multiplication to perform net slope correction for floating-point to fixed-point casts.
`UseDivisionForNetSlopeCompu` (Fixed-Point Designer) **off**, on, `Use division for reciprocals of integers only`	**Optimization** > **Use division for fixed-point net slope computation**	Perform net slope computation using division when simplicity and accuracy conditions are met.

Command-Line Information: Optimization Pane: Signals and Parameters Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`BufferReuse` off, **on**	**Optimization** > **Signals and Parameters** > **Reuse local block outputs**	Reuse local (function) variables for block outputs wherever possible. Selecting this option trades code traceability for code efficiency.
`EnableMemcpy` off, **on**	**Optimization** > **Signals and Parameters** > **Use memcpy for vector assignment**	Optimize code generated for vector assignment by replacing `for` loops with `memcpy` function calls.
`EnhancedBackFolding` **off**, on	**Optimization** > **Signals and Parameters** > **Minimize data copies between local and global variables**	Reuse existing global variables to store temporary results.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ExpressionFolding off, **on**	**Optimization > Signals and Parameters > Eliminate superfluous local variables (Expression folding) > Interface**	Collapse block computations into single expressions wherever possible. This improves code readability and efficiency.
InlineInvariantSignals off, **on**	**Optimization > Signals and Parameters > Inline invariant signals**	Precompute and inline the values of invariant signals in the generated code.
LocalBlockOutputs off, **on**	**Optimization > Signals and Parameters > Enable local block outputs**	Declare block outputs in local (function) scope wherever possible to reduce global RAM usage.
MemcpyThreshold int - **64**	**Optimization > Signals and Parameters > Memcpy threshold (bytes)**	Specify the minimum array size in bytes for which memcpy function calls should replace for loops in the generated code for vector assignments.
RollThreshold int - **5**	**Optimization > Signals and Parameters > Loop unrolling threshold**	Specify the minimum signal width for which a for loop is to be generated.
MaxStackSize <Specify a value>, **Inherit from target**	**Optimization > Signals and Parameters > Maximum stack size (bytes)**	Specify the maximum stack size in bytes for your model.

Command-Line Information: Optimization Pane: Stateflow Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`DataBitsets` (Stateflow) **off**, `on`	**Optimization > Stateflow > Use bitsets for storing Boolean data**	Use bit sets for storing Boolean data.
`StateBitsets` (Stateflow) **off**, `on`	**Optimization > Stateflow > Use bitsets for storing state configuration**	Use bit sets for storing state configuration.

Command-Line Information: Code Generation Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CheckMdlBeforeBuild` *string* - **off**, `warning`, `error`	**Code Generation > Check model before generating code**	Specify whether to run Code Generation Advisor checks before generating code.
`GenCodeOnly` *string* - **off**, `on`	**Code Generation > Generate code only**	Generate source code, but do not execute the makefile to build an executable.
`GenerateMakefile` *string* - `off`, **on**	**Code Generation > Generate makefile**	Specify whether to generate a makefile during the build process for a model.
`MakeCommand` *string* - **make_rtw**	**Code Generation > Make command**	Specify the `make` command and optional arguments to be used to generate an executable for the model.
`ObjectivePriorities` (GRT) string - **{''}**, `{'Debugging'}`, `{'Execution efficiency'}`	**Code Generation > Select objective**	Specify the code generation objectives to use with the Code Generation Advisor.
`ObjectivePriorities` (ERT) string - **{''}**, `{'Efficiency'}`, `{'Traceability'}`, `{'Safety precaution'}`, `{'Debugging'}`	**Code Generation > Set Objectives**	Specify and prioritize the code generation objectives to use with the Code Generation Advisor.
`RTWCompilerOptimization` string - **Off**, `On`, `Custom`	**Code Generation > Compiler optimization level**	Use this parameter to trade off compilation time against run time for your model code without having to

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		supply compiler-specific flags to other levels of the Simulink Coder build process. `Off` - Turn compiler optimizations off for faster builds `On` - Turn compiler optimizations on for faster code execution `Custom` - Specify custom compiler optimization flags via the `RTWCustomCompilerOptimizations` parameter
RTWCustomCompiler Optimizations string - `''`, unquoted string of compiler optimization flags	**Code Generation > Custom compiler optimization flags**	If you specified `Custom` to the `RTWCompilerOptimization` parameter, use this parameter to specify custom compiler optimization flags, for example, `-O2`.
SaveLog **off**, on	**Code Generation > Save build log**	Save build log.
SystemTargetFile string - **grt.tlc**	**Code Generation > System target file**	Specify a system target file.
TargetLang string - **C**, C++	**Code Generation > Language**	Specify whether to generate C or C++ code.
TemplateMakefile *string* - **grt_default_tmf**	**Code Generation > Template makefile**	Specify the current template makefile for building a Simulink Coder target.

Command-Line Information: Code Generation Pane: Report Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
GenerateReport string - **off**, on	**Code Generation > Report > Create code generation report**	Document the generated C or C++ code in an HTML report.
LaunchReport string - **off**, on	**Code Generation > Report > Launch report automatically**	Display the HTML report after code generation completes.

Command-Line Information: Code Generation Pane: Comments Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ForceParamTrailComments *string* - **off**, on	**Code Generation > Comments > Verbose comments for SimulinkGlobal storage class**	Specify that comments be included in the generated file. To reduce file size, the model parameters data structure is not commented when there are more than 1000 parameters.
GenerateComments *string* - off, **on**	**Code Generation > Comments > Include comments**	Include comments in generated code.
OperatorAnnotations *string* - **off**, on	**Code Generation > Comments > Operator annotations**	Specify whether to include operator annotations in the generated code as comments.
ShowEliminatedStatement *string* - **off**, on	**Code Generation > Comments > Show eliminated blocks**	Show statements for eliminated blocks as comments in the generated code.
SimulinkBlockComments *string* - off, **on**	**Code Generation > Comments > Simulink block / Stateflow object comments**	Insert Simulink block and Stateflow object names as comments above the generated code for each block.

Command-Line Information: Code Generation Pane: Symbols Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
MaxIdLength int - **31**	**Code Generation > Symbols > Maximum identifier length**	Specify the maximum number of characters that can be used in generated function, type definition, and variable names.
ReservedNameArray *string array* - **{}**	**Code Generation > Symbols > Reserved names**	Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code to avoid name conflicts.
UseSimReservedNames	**Code Generation > Symbols > Use the**	Specify whether to use the same reserved names as those specified in

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
string - **off**, on	**same reserved names as Simulation Target**	the **Simulation Target** > **Symbols** pane.

Command-Line Information: Code Generation Pane: Custom Code Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CustomHeaderCode *string* - ''	**Code Generation > Custom Code > Header file**	Specify code to appear near the top of the generated model header file.
CustomInclude *string* - ''	**Code Generation > Custom Code > Include directories**	Specify a space-separated list of include folders to add to the include path when compiling the generated code. **Note:** If your list includes Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, `'C:\Project "C:\Custom Files"'`
CustomInitializer *string* - ''	**Code Generation > Custom Code**	Specify code to appear in the generated model initialize function.
CustomLibrary *string* - ''	**Code Generation > Custom Code > Initialize function Libraries**	Specify a space-separated list of static library files to link with the generated code.
CustomSource *string* - ''	**Code Generation > Custom Code > Source files**	Specify a space-separated list of source files to compile and link with the generated code.
CustomSourceCode *string* - ''	**Code Generation > Custom Code > Source file**	Specify code to appear near the top of the generated model source file.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CustomTerminator *string* - ''	**Code Generation > Custom Code > Terminate function**	Specify code to appear in the generated model terminate function.
RTWUseSimCustomCode string - **off**, on	**Code Generation > Custom Code > Use the same custom code settings as Simulation Target**	Specify whether to use the same custom code settings as those in the **Simulation Target > Custom Code** pane.

Command-Line Information: Code Generation Pane: Debug Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ProfileTLC string - **off**, on	**Code Generation > Debug > Profile TLC**	Profile the execution time of each TLC file used to generate code for this model in HTML format.
RTWVerbose string - off, **on**	**Code Generation > Debug > Verbose build**	Display messages indicating code generation stages and compiler output.
RetainRTWFile string - **off**, on	**Code Generation > Debug > Retain .rtw file**	Retain the *model*.rtw file in the current build folder.
TLCAssert string - **off**, on	**Code Generation > Debug > Enable TLC assertion**	Produce a TLC stack trace when the argument to the assert directives evaluates to false.
TLCCoverage string - **off**, on	**Code Generation > Debug > Start TLC coverage when generating code**	Generate .log files containing the number of times each line of TLC code is executed during code generation.
TLCDebug string - **off**, on	**Code Generation > Debug > Start TLC debugger when generating code**	Start the TLC debugger during code generation at the beginning of the TLC program. TLC breakpoint statements automatically invoke the TLC debugger regardless of this setting.

Command-Line Information: Code Generation Pane: Interface Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CodeInterfacePackaging` string - `C++ class`, `Nonreusable function`, `Reusable function` (Default is `Nonreusable function` if `TargetLang` is set to `C`, or `C++ class` if `TargetLang` is set to `C++`)	**Code Generation > Interface > Code interface packaging**	Specify the packaging for the generated C or C++ code interface.
`CodeReplacementLibrary` string - **None**, `GNU C99 extensions`, `Intel IPP for x86-64 (Windows)`, `Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)`, `Intel IPP for x86/Pentium (Windows)`, `Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)`, `Intel IPP for x86-64 (Linux)`, `Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)` (For ERT-based models, additional values may be available; see the **Code replacement library** drop-down list in the Configuration Parameters dialog box.)	**Code Generation > Interface > Code replacement library**	Specify an application-specific math library for your model. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur. `None` `GNU C99 extensions` - GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99` `Intel IPP for x86-64 (Windows)` - Generates calls to the Intel Performance Primitives (IPP) library for the x86-64 Windows platform. `Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)` - GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions. `Intel IPP for x86/Pentium (Windows)` - Generates calls to the Intel Performance Primitives (IPP) library for the x86/Pentium Windows platform.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		`Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)` - Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86/Pentium Windows platform. `Intel IPP for x86-64 (Linux)` - Generates calls to the Intel Performance Primitives (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 Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE), with GNU C99 extensions, for the x86-64 Linux platform.
`CombineOutputUpdateFcns` string - `off`, **`on`**	**Code Generation > Interface > Single output/update function**	Generate a model's output and update routines into a single-step function.
`ExtMode` **`off`**, `on`	**Code Generation > Interface > Interface**	Specify the data interface to be generated with the code.
`ExtModeMexArgs` *string* `('')`	**Code Generation > Interface > Interface > External mode > MEX-file arguments**	Specify arguments that are passed to an external mode interface MEX-file for communicating with executing targets.
`ExtModeStaticAlloc` **`off`**, `on`	**Code Generation > Interface > Static memory allocation**	Use a static memory buffer for external mode instead of allocating dynamic memory (calls to `malloc`).

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ExtModeStaticAllocSize *integer* (**1000000**)	**Code Generation > Interface > Static memory buffer size**	Specify the size in bytes of the external mode static memory buffer.
ExtModeTransport int - **0** for TCP/IP, 1 for serial	**Code Generation > Interface > Interface > External mode > Transport layer**	Specify transport protocols for external mode communications.
GenerateASAP2 **off**, on	**Code Generation > Interface > Interface**	Specify the data interface to be generated with the code.
GRTInterface string - **off** (except on for GRT models created before R2012a), on	**Code Generation > Interface > Classic call interface**	Include a code interface (wrapper) that is compatible with the pre-R2012a GRT target.
LogVarNameModifier string - **none**, rt_, _rt	**Code Generation > Interface > MAT-file variable name modifier**	Augment the MAT-file variable name.
MatFileLogging string - off, on (Default is on for GRT targets, off for ERT targets)	**Code Generation > Interface > MAT-file logging**	Generate code that logs data to a MAT-file.
MultiInstanceErrorCode string - None, Warning, **Error**	**Code Generation > Interface > Multi-instance code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of multi-instance code.
RTWCAPIParams string - **off**, on	**Code Generation > Interface > Generate C API for: parameters**	Generate C API parameter tuning structures.
RTWCAPIRootIo string - **off**, on	**Code Generation > Interface > Generate C API for: root-level I/O**	Generate a C API root-level I/O structure

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
RTWCAPISignals string - **off**, on	**Code Generation > Interface > Generate C API for: signals**	Generate C API signal structure.
RTWCAPIStates string - **off**, on	**Code Generation > Interface > Generate C API for: states**	Generate C API state structure.
SupportNonFinite string - off, **on**	**Code Generation > Interface > Support non-finite numbers**	Support nonfinite values (inf, nan, -inf) in the generated code.
TargetLangStandard string - **C89/C90 (ANSI)**, C99 (ISO), C++03 (ISO)	**Code Generation > Interface > Standard math library**	Specify a standard math library for your model. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur. C89/C90 (ANSI) - ISO/IEC 9899:1990 C standard math library for floating-point functions C99 (ISO) - ISO/IEC 9899:1999 C standard math library C++03 (ISO) - ISO/IEC 14882:2003 C++ standard math library
UtilityFuncGeneration string - **Auto**, Shared location	**Code Generation > Interface > Shared code placement**	Specify where utility code is to be generated.

Command-Line Information: Not in GUI

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CodeGenDirectory	Not available	For MathWorks use only.
Comment	Not available	For MathWorks use only.
CompOptLevelCompliant off, on	Not available	Set in SelectCallback for a target to indicate whether the target supports the ability to use the **Compiler optimization**

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		level parameter on the **Code Generation** pane to control the compiler optimization level for building generated code. Default is `off` for custom targets and `on` for targets provided with the Simulink Coder and Embedded Coder products.
`ConfigAtBuild`	Not available	For MathWorks use only.
`ConfigurationMode`	Not available	For MathWorks use only.
`ConfigurationScript`	Not available	For MathWorks use only.
`ERTCustomFileBanners`	Not available	For MathWorks use only.
`EvaledLifeSpan`	Not available	For MathWorks use only.
`ExtModeMexFile`	Not available	For MathWorks use only.
`ExtModeTesting`	Not available	For MathWorks use only.
`FoldNonRolledExpr`	Not available	For MathWorks use only.
`GenerateFullHeader`	Not available	For MathWorks use only.
`GenerateSharedConstants`	Not available	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.
`IncAutoGenComments`	Not available	For MathWorks use only.
`IncludeRegionsInRTWFile BlockHierarchyMap`	Not available	For MathWorks use only.
`IncludeRootSignalInRTWFile`	Not available	For MathWorks use only.
`IncludeVirtualBlocksInRTW FileBlockHierarchyMap`	Not available	For MathWorks use only.
`IsERTTarget`	Not available	For MathWorks use only.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ModelReferenceCompliant` string - `off`, **on**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports model reference.
`ParamNamingFcn`	Not available	For MathWorks use only.
`PostCodeGenCommand` *string* - `''`	Not available	Add the specified post code generation command to the model build process.
`PreserveName`	Not available	For MathWorks use only.
`PreserveNameWithParent`	Not available	For MathWorks use only.
`ProcessScript`	Not available	For MathWorks use only.
`ProcessScriptMode`	Not available	For MathWorks use only.
`SignalNamingFcn`	Not available	For MathWorks use only.
`SystemCodeInlineAuto`	Not available	For MathWorks use only.
`TargetFcnLib`	Not available	For MathWorks use only.
`TargetLibSuffix` *string* - `''`	Not available	Control the suffix used for naming a target's dependent libraries (for example, `_target.lib` or `_target.a`). If specified, the string 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` *string* - `''`	Not available	Control the location of precompiled libraries. If you do not set this parameter, the code generator uses the location specified in `rtwmakecfg.m`.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TargetPreprocMaxBitsSint` int - **32**	Not available	Specify the maximum number of bits that the target C preprocessor can use for signed integer math.
`TargetPreprocMaxBitsUint` int - **32**	Not available	Specify the maximum number of bits that the target C preprocessor can use for unsigned integer math.
`TargetTypeEmulationWarn SuppressLevel SuppressLevel` int - **0**	Not available	When greater than or equal to 2, suppress warning messages that the Simulink Coder software displays when emulating integer sizes in rapid prototyping environments.
`TLCOptions` *string* - `''`	Not available	Specify additional TLC command line options.

Model Advisor Checks

Simulink Coder Checks

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

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 model solver and sample time configuration settings.

Description

Incorrect configuration settings can stop the Simulink Coder software from generating code. Underspecifying sample times can lead to undesired results. Avoid generating code that might corrupt data or produce unpredictable behavior.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The solver type is set incorrectly for model level code generation.	In the Configuration Parameters dialog box, on the **Solver** pane, set • "**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 Embedded Coder build process automatically changes **Solver type** to `fixed-step` when you select **Code Generation** > **Build Subsystem** or **Code Generation** > **Generate S-Function** from the subsystem context menu.

See Also

• "Configure Scheduling"

- "Execute Multitasking Models"

Check for blocks not supported by code generation

Identify blocks not supported by code generation.

Description

This check partially identifies model constructs that are not suited for code generation as identified in the Simulink Block Support tables for Simulink Coder and Embedded Coder. If you are using blocks with support notes for code generation, review the information and follow the given advice.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for code generation.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.

Capabilities and Limitations

You can:

- Run this check on your library models.
- 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 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"
- "Build configuration"

The template makefile approach uses the following settings to build generated code:

- **Compiler optimization level**
- **Custom compiler optimization flags**
- **Generate makefile**
- **Template makefile**

Results and Recommended Actions

Condition	Recommended Action	Comment
Model is configured to use the toolchain approach.	No action.	The model was automatically upgraded.
Model is not configured to use the toolchain approach.	Model cannot be automatically upgraded to use the toolchain approach.	The system target file is not toolchain-compliant. Set **System target file** to a toolchain-compliant target, such as `ert.tlc`, `ert_shrlib.tlc`, or `grt.tlc`.

Condition	Recommended Action	Comment
Model is not configured to use the toolchain approach. (Parameter values are not the default values.)	Model can be automatically upgraded to use the toolchain approach. Click **Update Model**.	The parameters are set to their default values, except **Compiler Optimization Level**, which is set `Optimizations on (faster runs)`. Clicking **Update Model** sets **Compiler Optimization Level** to its default value, `Optimizations off (faster builds)`, and then upgrades the model. The upgraded model has **Build Configuration** set to `Faster Builds`. Saving the model makes these changes permanent.
Model is not configured to use the toolchain approach. (Parameter values are not the default values.)	Model cannot be automatically upgraded to use the toolchain approach.	One or more of the following parameters is not set to its default value: • **Generate makefile** (default: Enabled) • **Template makefile** (default: Target-specific default TMF) • **Compiler optimization level** (default: `Optimizations off (faster builds)`) • **Make command** (default: `make_rtw` without arguments) See "Upgrade Model to Use Toolchain Approach"

Action Results

Clicking **Update model** upgrades the model to use the toolchain approach to build generated code.

See Also

• "Upgrade Model to Use Toolchain Approach"

Check and update the embedded target model to use ert.tlc system target file

Check and update the embedded target model to use ert.tlc system target file.

Description

Check and update models whose **System target file** is set to `idelink_ert.tlc` or `idelink_grt.tlc` and whose target hardware is one of the supported Texas Instruments C2000™ processors to use `ert.tlc` and similar settings.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
System target file is set to `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"

Check for blocks that have constraints on tunable parameters

Identify blocks with constraints on tunable parameters.

Description

Lookup Table blocks have strict constraints when they are tunable. If you violate lookup table block restrictions, the generated code produces wrong answers.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: • 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**

Condition	Recommended Action
	input values, and **Vector of output values** parameters if you specify multiple zero values for the **Row index input values** or **Column index of input values** parameters.
Lookup Table (n-D) blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must preserve the increasing monotonicity of the breakpoint values for each table dimension **Breakpoints n**.

Capabilities and Limitations

If you have a Simulink 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

Identify referenced model configuration parameter settings that do not match the top model configuration parameter settings.

Description

The code generator cannot create code for top models that contain referenced models with different, incompatible configuration parameter settings.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The top model and the referenced model have inconsistent model configuration parameter settings.	Modify the specified model configuration settings.

See Also

Model Referencing Configuration Parameter Requirements

Check sample times and tasking mode

Set up the sample time and tasking mode for your system.

Description

Incorrect tasking mode can result in inefficient code execution or incorrect generated code.

Available with Simulink Coder.

Results and Recommended Actions

Condition	Recommended Action
The model represents a multirate system but is not configured for multitasking.	In the Configuration Parameters dialog box, on the **Solver** pane, set the "**Tasking mode for periodic sample times**" parameter as recommended.
The model is configured for multitasking, but multitasking is not desirable on the target hardware.	In the Configuration Parameters dialog box, on the **Solver** pane, set the "**Tasking mode for periodic sample times**" parameter to `SingleTasking`, or change the settings on the "**Hardware Implementation**" pane.

See Also

"Single-Tasking and Multitasking Execution Modes"

Code Generation Advisor Checks

- "Available Checks for Code Generation Objectives" on page 5-11
- "Identify questionable blocks within the specified system" on page 5-16
- "Check model configuration settings against code generation objectives" on page 5-17

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:2004 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"Check model configuration settings against code generation objectives" on page 5-17	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 5-16	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 is available	N/A	N/A	N/A	N/A	N/A	N/A
"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	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 precautions (ERT-based targets)	Traceability (ERT-based targets)	Debugging (all targets)	MISRA-C:2004 guidelines (ERT-based targets)	Polyspace (ERT-based targets)
		is available						
"Identify questionable code instrumentation (data I/O)"	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	Included when Traceability or Debugging are not higher priorities and Embedded Coder is available	N/A	N/A	N/A	N/A	N/A
"Identify questionable subsystem settings"	N/A	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A
"Identify blocks that generate expensive rounding code"	Included if Embedded Coder is available	Included if Embedded Coder is available	N/A	N/A	N/A	N/A	N/A	N/A

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:2004 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"Identify questionable fixed-point operations"	Included if Embedde Coder or Fixed-Point Designer is available	Included if Embedde Coder or Fixed-Point Designer is available	N/A	N/A	N/A	N/A	N/A	N/A
"Identify blocks using one-based indexing" on page 5-2	Included	Included	N/A	N/A	N/A	N/A	N/A	N/A
"Identify lookup table blocks that generate expensive out-of-range checking code"	Included if Embedde Coder is available	Included if Embedde 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
"Identify unconnected lines, input ports, and output ports"	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:2004 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"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 5-9	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 for partial 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:2004 guideline (ERT-based targets)	Polyspace (ERT-based targets)
"Check for blocks not recommended for MISRA-C:2004 compliance"	N/A	N/A	N/A	N/A	N/A	N/A	Included if Embedd Coder is available	N/A

See Also

- "Application Objectives" in the Simulink Coder documentation.
- "Application Objectives" 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 5-2.
- Simulink Verification and Validation Model Advisor Check Reference in the Simulink Verification and Validation documentation.

Identify questionable blocks within the specified system

Identify blocks not supported by code generation or not recommended for deployment.

Description

The code generator creates code only for the blocks that it supports. Some blocks are not recommended for production code deployment.

Results and Recommended Actions

Condition	Recommended Action
A block is not supported by the Simulink Coder software.	Remove the specified block from the model or replace the block with the recommended block.

Condition	Recommended Action
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

- The Simulink Coder "Recommended Settings Summary" on page 4-224
- The Embedded Coder "Recommended Settings Summary"
- "Application Objectives" in the Simulink Coder documentation.
- "Application Objectives" in the Embedded Coder documentation.

6

Parameters for Creating Protected Models

Create Protected Model
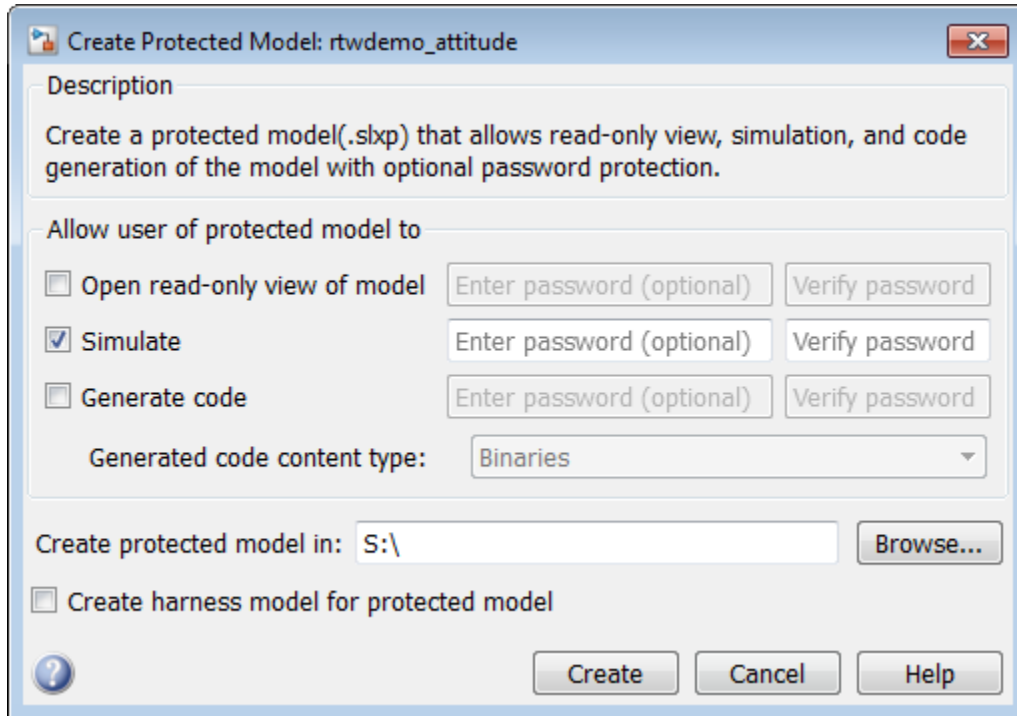
Create Protected Model: Overview

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"

Generate code

Allows user to generate code for the protected model with optional password protection. Selecting **Generate code**:

- Enables Simulation Report and Code Generation Report for the protected model.
- Sets Mode to enable code generation.
- Enables support for simulation.
- Displays code in the build folder in obfuscated form.

Settings

Default: Off

☑ 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 **Generated code content type**.

Alternatives

"Simulink.ModelReference.protect"

See Also

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

Generated code content type

Select the appearance of the generated code. When you select the **Generate code** check box, this parameter is enabled.

Settings

Default: `Obfuscated source code`

`Binaries`

    Includes only binaries for the generated code.

`Obfuscated source code`

    Includes obfuscated headers and binaries for the generated code.

`Readable source code`

Includes readable source code.

Dependencies

This parameter is enabled by selecting the **Generate code** check box.

Alternatives

"Simulink.ModelReference.protect"

See Also

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

Create protected model in

Specify the folder path for the protected model.

Settings

Default: Current working folder

Dependencies

A model that you protect must be available on the MATLAB path.

Alternatives

"Simulink.ModelReference.protect"

See Also

- "Protect a Referenced Model"
- "Create a Protected Model"

Create harness model for protected model

Create a harness model for the protected model. The harness model contains only a Model block that references the protected model.

Settings

Default: Off

☑ On

   Create a harness model for the protected model.

☐ Off

   Do not create a harness model for the protected model.

Alternatives

"Simulink.ModelReference.protect"

See Also

- "Harness Model"
- "Test the Protected Model"

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 your are developing 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
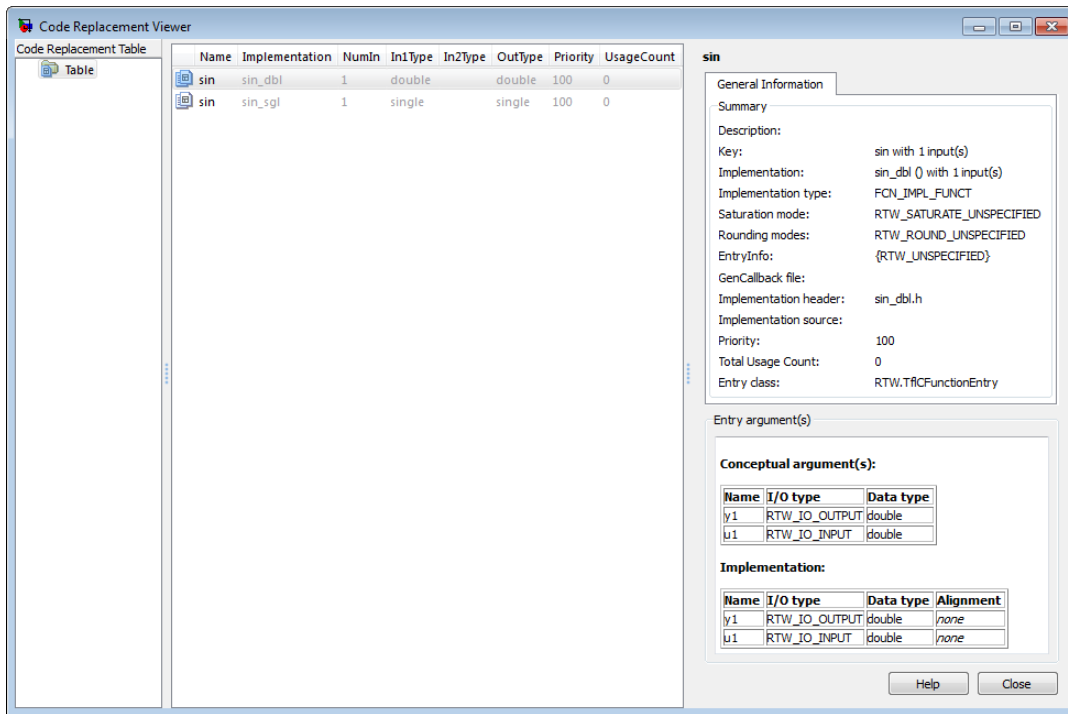
Open from the MATLAB command prompt using `RTW.viewTfl`

Examples

Display Contents of Code Replacement Library

```
RTW.viewTfl('Sin Function Example')
```
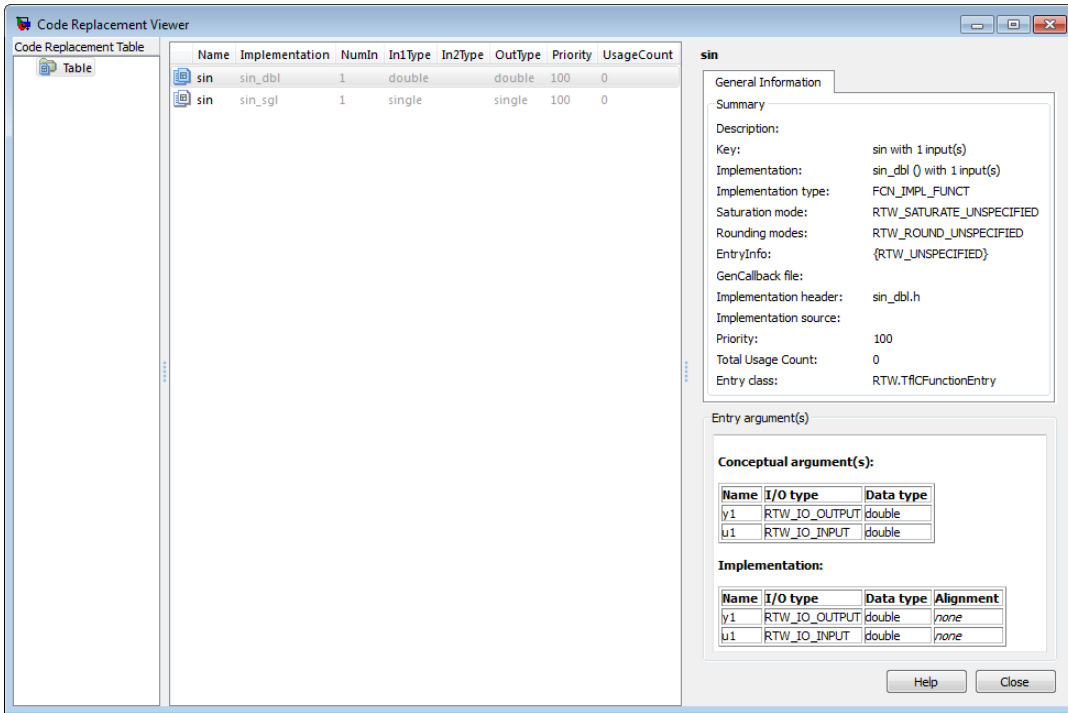
Display Contents of Code Replacement Table

```
RTW.viewTfl(crl_ttable_sinfcn)
```

- "Choose a Code Replacement Library"

Programmatic Use

`RTW.viewTfl(library)` opens the Code Replacement Viewer and displays the contents of library, where library is a string that names a registered code replacement library. For example, `'GNU 99 extensions'`.

`RTW.viewTfl(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"