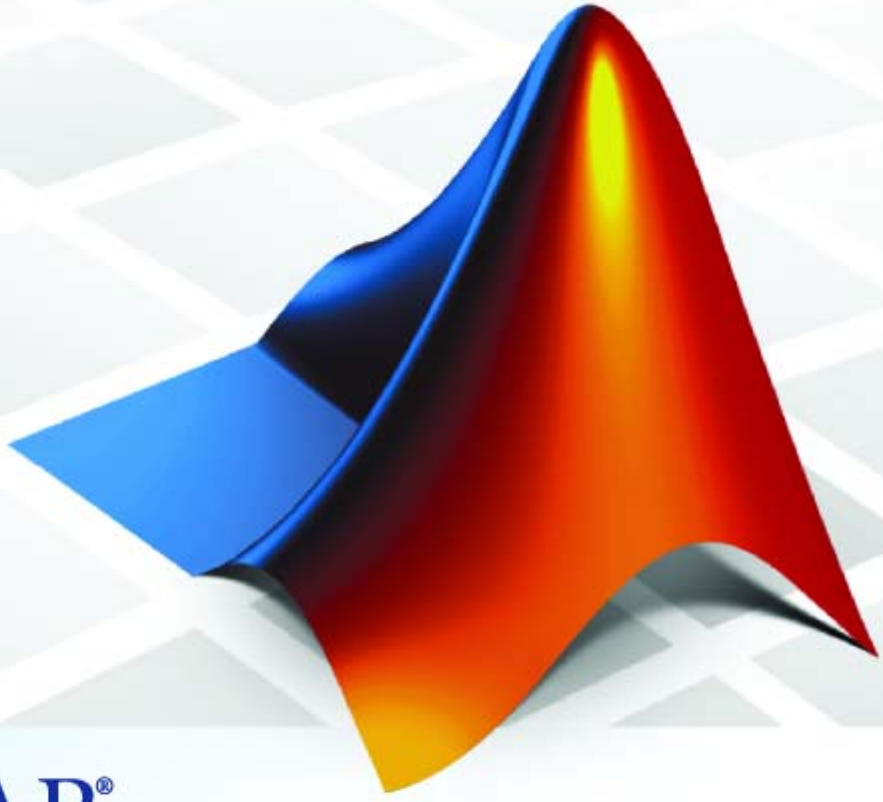
Real-Time Workshop® 7

Reference



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Real-Time Workshop Reference

Revision History

March 2006	Online only	New for Version 6.4
September 2006	Online only	Revised for Version 6.5 (Release 2006b)
March 2007	Online only	Revised for Version 6.6 (Release 2007a)
September 2007	Online only	Revised for Version 7.0 (Release 2007b)

Contents

1 Functions — By Category

2 Functions — Alphabetical List

3 Simulink Block Support

4 Blocks — By Category

5 Blocks — Alphabetical List

6 Configuration Parameters

7 Embedded MATLAB Coder Configuration Parameters

8 Model Advisor Checks

Index

1

Functions — By Category

Build Information

`addCompileFlags`	Add compiler options to model's build information
`addDefines`	Add preprocessor macro definitions to model's build information
`addIncludeFiles`	Add include files to model's build information
`addIncludePaths`	Add include paths to model's build information
`addLinkFlags`	Add link options to model's build information
`addLinkObjects`	Add link objects to model's build information
`addSourceFiles`	Add source files to model's build information
`addSourcePaths`	Add source paths to model's build information
`findIncludeFiles`	Find and add include (header) files to build information object
`getCompileFlags`	Compiler options from model's build information
`getDefines`	Preprocessor macro definitions from model's build information
`getIncludeFiles`	Include files from model's build information
`getIncludePaths`	Include paths from model's build information
`getLinkFlags`	Link options from model's build information
`getSourceFiles`	Source files from model's build information

`getSourcePaths`	Source paths from model's build information
`packNGo`	Package model code in zip file for relocation
`updateFilePathsAndExtensions`	Update files in model's build information with missing paths and file extensions
`updateFileSeparator`	Change file separator used in model's build information

Embedded MATLAB Coder

`emlc`	Generate C code or C MEX code directly from M-code

Project Documentation

`rtwreport`	Document generated code

Rapid Simulation

`rsimgetrtp`	Model's global parameter structure

Target Language Compiler Library

See the "TLC Function Library Reference" in the Real-Time Workshop Target Language Compiler documentation.

2

Functions — Alphabetical List

addCompileFlags

Purpose

Add compiler options to model's 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 a compiler option
- A single group name to multiple compiler options
- Multiple group names to collections of compiler options

To...	Specify *groups* as a...
Apply one group name to all compiler options	Character array. To specify compiler options to be used in the standard Real-Time Workshop makefile build process, specify the character array `'OPTS'` or `'OPT_OPTS'`.
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*. Available for nonmakefile build environments only.

Note To control compiler optimizations for your Real-Time Workshop makefile build at Simulink GUI level, use the **Compiler optimization level** option on the **Real-Time Workshop** pane of the Simulink Configuration Parameters dialog box. The **Compiler optimization level** option 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 specify compiler options for your Real-Time Workshop 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's build information. Real-Time Workshop 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 `MemOpt`.

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

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

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

- Add the compiler options `-Zi`, `-Wall`, and `-O3` to build information `myModelBuildInfo`. Place the options `-Zi` and `-Wall` in the group `Debug` and option `-O3` in the group `MemOpt`.

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

See Also

`addDefines`, `addLinkFlags`
"Programming a Post Code Generation Command"

addDefines

Purpose

Add preprocessor macro definitions to model's 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 an macro definition
- A single group name to multiple macro definitions
- Multiple group names to collections of multiple macro definitions

To...	Specify *groups* as a...
Apply one group name to all macro definitions	Character array. To specify macro definitions to be used in the standard Real-Time Workshop makefile build process, specify the character array `'OPTS'` or `'OPT_OPTS'`.
Apply different group names to macro definitions	Cell array of character arrays such that the number of group names matches the number elements you specify for *`macrodefs`*. Available for nonmakefile build environments only.

Description

The `addDefines` function adds specified preprocessor macro definitions to the model's build information. Real-Time Workshop 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 `Release`.

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

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

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

- Add the compiler definitions `-DPROTO`, `-DDEBUG`, and `-DPRODUCTION`, to build information `myModelBuildInfo`. Group the definitions `-DPROTO` and `-DDEBUG` with the string `Debug` and the definition `-DPRODUCTION` with the string `Release`.

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

See Also

`addCompileFlags`, `addLinkFlags`
"Programming a Post Code Generation Command"

Purpose

Add include files to model's 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 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 all 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's build information. Real-Time Workshop 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 all include files it adds to the build information
Cell array of character arrays	Pairs each character array with a specified include file. Thus, the length of the cell array must match the length of the cell array you specify for *filenames*.

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

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', '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'}, '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'},...
{'AppFiles' 'AppFiles' 'SysFiles'});
```

See Also

`addIncludePaths`, `addSourceFiles`, `addSourcePaths`, `updateFilePathsAndExtensions`, `updateFileSeparator`
"Programming a Post Code Generation Command"

addIncludePaths

Purpose

Add include paths to model's 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 all 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's build information. Real-Time Workshop 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 all 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'});
```

See Also

`addIncludeFiles`, `addSourceFiles`, `addSourcePaths`, `updateFilePathsAndExtensions`, `updateFileSeparator`
"Programming a Post Code Generation Command"

Purpose

Add link options to model's 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 a compiler option
- A single group name to multiple compiler options
- Multiple group names to collections of multiple compiler options

To...	Specify *groups* as a...
Apply one group name to all linker options	Character array. To specify linker options to be used in the standard Real-Time Workshop makefile build process, specify the character array `'OPTS'` or `'OPT_OPTS'`.
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*. Available for nonmakefile build environments only.

Description

The `addLinkFlags` function adds specified linker options to the model's build information. Real-Time Workshop 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 `Temp`.

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

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

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

- Add the linker options `-MD`, `-Gy`, and `-T` to build information `myModelBuildInfo`. Place the options `-MD` and `-Gy` in the group `Debug` and the option `-T` in the group `Temp`.

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

See Also

`addCompileFlags`, `addDefines`
"Programming a Post Code Generation Command"

addLinkObjects

Purpose

Add link objects to model's build information

Syntax

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

All arguments except *buildinfo* , *linkobjs*, and *paths* are optional. If you specify an optional argument, you must specify all of 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.

linkonly (optional)
: The logical value `true` or `false` or a vector of logical values that indicates whether each specified link object is to be only linked. If you set this argument to `false`, the function also adds a rule to the makefile for building the objects.

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* a...
Apply one group name to all 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 elements you specify for *linkobjs*.

Description

The `addLinkObjects` function adds specified link objects to the model's build information. Real-Time Workshop 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 all 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 all objects it adds to the build information.
Vector of values	Pairs each value with a specified object. Thus, the length of the vector must match the length of the cell array you specify for *`linkobjs`*.

If you choose to specify an optional argument, you must specify all of the optional arguments preceding it. For example, to specify that all 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 linkable. Since 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'}, [26 10],...
  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, but not linkable, and group them `MyTest`.

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

See Also "Programming a Post Code Generation Command"

Purpose

Add source files to model's 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 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 all 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's build information. Real-Time Workshop 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 all 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*.

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', '',...
'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'}, '', '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'}, '',...
{'Tests' 'Tests' 'Drivers'});
```

See Also

`addIncludeFiles`, `addIncludePaths`, `addSourcePaths`, `updateFilePathsAndExtensions`, `updateFileSeparator`
"Programming a Post Code Generation Command"

addSourcePaths

Purpose

Add source paths to model's 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 Real-Time Workshop 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 all 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's build information. Real-Time Workshop 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 all 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 Real-Time Workshop does not check whether a specified path string is valid.

Examples

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

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

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

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

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

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

See Also

`addIncludeFiles`, `addIncludePaths`, `addSourceFiles`, `updateFilePathsAndExtensions`, `updateFileSeparator`
"Programming a Post Code Generation Command"

Purpose

Generate C code or C MEX code directly from M-code

Syntax

```
emlc [-options] [files] fcn
```

Description

`emlc` invokes the Embedded MATLAB Coder from the MATLAB command prompt.

`emlc [-options] [files] fcn` translates the M-file *fcn*`.m` to a C MEX file or to embeddable C code, depending on the target you specify as an option on the command line (see "-T Specify Target" on page 2-36 in "Options" on page 2-29). If you generate embeddable C code, you can specify custom *files* to include in the build, as described in "Specifying Custom C Files on the Command Line" in the Real-Time Workshop documentation.

By default, `emlc` *fcn* does the following:

- Converts the M-function *fcn*`.m` to a C MEX function
- Generates a platform-specific MEX file in the current directory
- Generates the necessary wrapper files — such as C, header, object, and map files — in the subdirectory `emcprj/ mexfcn/`*fcn*`/`

You can change the default behavior by specifying one or more compilation *options* as described in "Options" on page 2-29.

Options

You can specify one or more compilation options with each `emlc` command. Use spaces to separate options and arguments. Embedded MATLAB Coder resolves options from left to right, so if you use conflicting options, the rightmost one prevails. Here is the list of `emlc` options:

- "-c Generate Code Only" on page 2-30
- "-d Specify Output Directory" on page 2-30
- "-eg Specify Input Properties by Example" on page 2-31

-c Generate Code Only

Generate code, but do not invoke the `make` command. Embedded MATLAB Coder does not compile the M-code or build a C code executable. Use this option only for `rtw`, `rtw:exe`, and `rtw:lib` targets (see "-T Specify Target" on page 2-36).

-d Specify Output Directory

`-d` *out_directory*

Store generated files in directory path specified by *out_directory*. If any directories on the path do not exist, Embedded MATLAB Coder creates them for you. *out_directory* can be an absolute path or relative path. If you do not specify an output directory, Embedded MATLAB Coder stores generated files in a default subdirectory:

```
emcprj/target/function
```

target represents the compilation target type, specified as follows:

Target Type	*target* Subdirectory
default	`mexfcn`
`-T MEX`	`mexfcn`
`-T RTW:EXE`	`rtwexe`
`-T RTW:LIB`	`rtwlib`

Note To specify a compilation target type, see "-T Specify Target" on page 2-36.

-eg Specify Input Properties by Example

`-eg` *example_inputs*

Use the values in cell array *example_inputs* as sample inputs for defining the properties of the primary M-function inputs. The cell array should provide the same number and order of inputs as the primary function. See "Defining Input Properties by Example at the Command Line" .

-F Specify Default fimath

`-F` *fimath*

Use *fimath* as the default `fimath` object for all fixed-point inputs to the primary function. You can define the default value using the Fixed-Point Toolbox `fimath` function, as in this example:

```
emlc -F fimath('OverflowMode','saturate','RoundMode','nearest')
```

Embedded MATLAB Coder uses the default value if you have not defined any other `fimath` property for the primary, fixed-point inputs, either by example (see "Defining Input Properties by Example at the Command Line") or programmatically (see "Defining Input Properties Programmatically in the M-File"). If you do not define a default value, then you must use one of the other methods to specify the `fimath` property of your primary, fixed-point inputs.

-g Compile C MEX Function in Debug Mode

Compile the C MEX function in debug mode, with optimization turned off. Applies only to C MEX generation. If you do not specify `-g`, `emlc` compiles the C MEX function in optimized mode. You specify these modes using the `mex -setup` procedure described in "Building MEX-Files" in the online MATLAB External Interfaces documentation.

-I Add Directories to Embedded MATLAB Path

`-I` *include_path*

Add *include_path* to the Embedded MATLAB path. By default, the Embedded MATLAB path consists of the current directory (`pwd`) and the Embedded MATLAB libraries directory. `emlc` searches the Embedded MATLAB path *first* when converting M-code to C code. See "File Paths and Naming Conventions".

-N Specify Default Numeric Type

`-N` *numerictype*

Use *numerictype* as the default `numerictype` object for all fixed-point inputs to the primary function. You can define the default value using the Fixed-Point Toolbox `numerictype` function, as in this example:

```
emlc -N numerictype(1,32,23) myFcn
```

This command specifies that the numeric type of all fixed-point inputs to the top-level function `myFcn` be signed (1), have a word length of 32, and have a fraction length of 23.

Embedded MATLAB Coder uses the default value if you have not specified any other `numerictype` for the primary, fixed-point inputs, either by example (see "Defining Input Properties by Example at the Command Line") or programmatically (see "Defining Input Properties Programmatically in the M-File"). If you do not define a default value, then you must use one of the other methods to specify the `numerictype` of your primary, fixed-point inputs.

-o Specify Output File Name

`-o` *output_file_name*

Generate the final output file—that is, the C MEX function, Real-Time Workshop executable, or Real-Time Workshop library— with the base name *output_file_name*. If the output file is a C MEX function, Embedded MATLAB Coder assigns it a platform-specific extension.

You can specify *output_file_name* as a file name or an existing path, with the following effects:

If you specify:	emlc:
A file name	Copies the MEX-file to the current directory
An existing path	Generates the MEX-file in the directory specified by the path, but does not copy the MEX-file to the current directory
A path that does not exist	Generates an error

Embedded MATLAB Coder generates the supporting C files with the same base name as the corresponding M-files, replacing the `.m` extension with `.c`.

-O Specify Compiler Optimization Option

`-O` *optimization_option*

Specify compiler *optimization_option* with one of the following literals (no quotes):

Compiler Optimization Option	Action
`disable:inline`	Disable function inlining.
`enable:inline`	Enable function inlining (default).

-report Generate Compilation Report

Generate a compilation report. If this option is not specified, `emlc` generates a report only if there are compilation messages. See "Working with Compilation Reports".

-s Specify Configuration Properties

`-s` *config_object*

Generate code based on the properties of configuration object *config_object*. When you specify conflicting configuration objects on the command line, the rightmost configuration object prevails. For detailed information about working with configuration objects, see "Configuring Your Environment for Code Generation".

If a configuration object is not specified, Embedded MATLAB Coder uses default property values, as follows:

Defaults for emlcoder.MEXConfig.

Property	Default
`Name`	`'Automatic C MEX Generation'`
`EnableDebugging`	`false`
`GenerateReport`	`false`
`LaunchReport`	`false`
`CustomSourceCode`	`''`
`CustomHeaderCode`	
`CustomInitializer`	
`CustomTerminator`	
`CustomInclude`	
`CustomSource`	
`CustomLibrary`	

Defaults for emlcoder.RTWConfig.

Property	Default
Name	'Real-Time Workshop'
RTWVerbose	false
GenCodeOnly	false
GenerateMakefile	true
GenerateReport	false
LaunchReport	false
MaxIDLength	31
GenFloatMathFcnCalls	'ANSI_C'
MakeCommand	'make_rtw'
TemplateMakeFile	'grt_default_tmf'
PostCodeGenCommand	''
CustomSourceCode	
CustomHeaderCode	
CustomInitializer	
CustomTerminator	
CustomInclude	
CustomSource	
CustomLibrary	

Defaults for emlcoder.HardwareImplementation.

Property	Default
Name	'Hardware Implementation'

Property	Default
ProdHWDeviceType	'Generic->MATLAB Host Computer'
ProdBitPerChar	8
ProdBitPerShort	16
ProdBitPerInt	32
ProdBitPerLong	32
ProdWordSize	32
ProdShiftRightIntArith	true
ProdEndianess	'LittleEndian'
ProdIntDivRoundTo	'Zero'

-T Specify Target

`-T` *target_option*

Specify a target option as follows:

Target Option	Action
`mex`	Generate a C MEX function (default).

`rtw`
or
`rtw:exe`
	Generate embeddable C code and compile it to an executable (`.exe` file).

`rtw:lib`	Generate embeddable C code and compile it to a library (`.lib` file).

Note The `rtw:exe`, `rtw`, and `rtw:lib` options require Real-Time Workshop.

See “Choosing Your Target”.

-v Show Compilation Steps

Enable verbose mode to show compilation steps.

-? Display Help

Display the`emlc` command help.

findIncludeFiles

Purpose

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 any 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 all source and include paths recorded in a model's build information object
- Adds the files found, along with their full paths, to the build information object
- Deletes duplicate entries

Examples

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

See Also

"Programming a Post Code Generation Command"

getCompileFlags

Purpose

Compiler options from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

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.

Returns

Compiler options stored in the model's build information.

Description

The getCompileFlags function returns compiler options stored in the model's 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 all compiler options stored in build information myModelBuildInfo.

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

```
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 all 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'
```

See Also

`getDefines`, `getLinkFlags`
"Programming a Post Code Generation Command"

getDefines

Purpose

Preprocessor macro definitions from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

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.

Returns

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

Vector	Description
macrodef	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's 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 all preprocessor macro definitions stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addDefines(myModelBuildInfo, {'PROTO=first' '-DDEBUG'...
'test' '-dPRODUCTION'}, {'Debug' 'Debug' 'Debug'...
'Release'});
[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 all preprocessor macro definitions stored in build information `myModelBuildInfo` except those with the group name `Debug`.

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

defs =

    '-DPRODUCTION'
```

See Also

`getCompileFlags`, `getLinkFlags`
"Programming a Post Code Generation Command"

getIncludeFiles

Purpose

Include files from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

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.

replaceMatlabroot
The logical value `true` or `false`.

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

Returns

Names of include files stored in the model's build information.

Description

The `getIncludeFiles` function returns the names of include files stored in the model's 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 all include paths and filenames stored in build information `myModelBuildInfo`.

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

incfiles =

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

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

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

incfiles =

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

See Also

`getIncludePaths`, `getSourceFiles`, `getSourcePaths`
"Programming a Post Code Generation Command"

Purpose

Include paths from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

replaceMatlabroot
The logical value `true` or `false`.

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

Returns

Paths of include files stored in the model's build information.

Description

The `getIncludePaths` function returns the names of include file paths stored in the model's 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 all 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''
```

See Also

`getIncludeFiles`, `getSourceFiles`, `getSourcePaths`
"Programming a Post Code Generation Command"

Purpose

Link options from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

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*.

Returns

Linker options stored in the model's build information.

Description

The `getLinkFlags` function returns linker options stored in the model's 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 all linker options stored in build information `myModelBuildInfo`.

```
myModelBuildInfo = RTW.BuildInfo;
addLinkFlags(myModelBuildInfo, {'-MD -Gy' '-T'},...
{'Debug' 'MemOpt'});
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 all compiler options stored in build information `myModelBuildInfo` except those with the group name `Debug`.

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

linkflags =

    '-T'
```

See Also

`getCompileFlags`, `getDefines`
"Programming a Post Code Generation Command"

getSourceFiles

Purpose

Source files from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

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.

replaceMatlabroot
The logical value `true` or `false`.

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

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.

Returns

Names of source files stored in the model's build information.

Description

The `getSourceFiles` function returns the names of source files stored in the model's 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 source files the function returns.

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

Examples

- Get all 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'
```

See Also

`getIncludeFiles`, `getIncludePaths`, `getSourcePaths`
"Programming a Post Code Generation Command"

Purpose

Source paths from model's build information

Syntax

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

Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

replaceMatlabroot
The logical value `true` or `false`.

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

Returns

Paths of source files stored in the model's build information.

Description

The `getSourcePaths` function returns the names of source file paths stored in the model's 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 all 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 directory in this case is `\\myserver\myworkspace\matlab`.

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

```
ans =

$(MATLAB_ROOT)\rtw\c\libsrc

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

ans =

\\myserver\myworkspace\matlab\rtw\c\libsrc
```

See Also

`getIncludeFiles`, `getIncludePaths`, `getSourceFiles`
"Programming a Post Code Generation Command"

Purpose

Package model code in zip file for relocation

Syntax

`packNGo(`*`buildinfo`*`, `*`propVals`*`...)`

`propVals` is optional.

Arguments

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

`propVals` (optional)
A cell array of property-value pairs that specify packaging details.

To...	Specify Property...	With Value...
Package all model code files in a zip file as a single, flat directory	`'packType'`	`'flat'` (default)
Package model code files hierarchically in a primary zip file that contains three secondary zip files: • `mlrFiles.zip` — files in your *`matlabroot`* directory tree • `sDirFiles.zip` — files in and under your build directory • `otherFiles.zip` — required files not in the *`matlabroot`* or `start` directory trees	`'packType'`	`'hierarchical'`Paths for files in the secondary zip files are relative to the root directory of the primary zip file.
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.

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`)
- Header files (for example, `.h` and `.hpp`)
- MAT-file that contains the model's build information object (`.mat`)

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 directory 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.

Examples

- Package the code files for model `zingbit` in the file `zingbit.zip` as a flat directory 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 zingbat_grt_rtw;
  load buildInfo.mat
  packNGo(buildInfo, {'packType', 'hierarchical', ...
   'fileName', 'portzingbit'});
  ```

See Also

"Programming a Post Code Generation Command"
"Relocating Code to Another Development Environment"

Purpose

Document generated code

Syntax

```
rtwreport(model, dir)
```

dir is optional.

Arguments

model
: The model for which generated code is to be documented.

dir (optional)
: The directory that contains the generated code. Specify this argument only if the build directory is not in the current directory or in the directory that stores the model. The directory you specify must be a standard build directory and its parent directory must include the model's project directory (`slprj`) .

Description

The `rtwreport` function generates a report that documents the code generated by Real-Time Workshop for a specified model. If necessary, the function loads the model and generates code before generating the report, which includes:

- Snapshots of block diagrams of the model and its subsystems
- Block execution order
- Summary of the generated code
- Full listings of the generated code that resides in the build directory

By default, Real-Time Workshop names the generated report `codegen.html` and places the file in the current directory. If you specify an optional directory, Real-Time Workshop places the file `codegen.html` in the parent directory of the specified directory. If the specified directory is not found, an error results and Real-Time Workshop does not attempt to generate code for the model.

Example

Generate a report for `mymodel`.

```
rtwreport(mymodel);
```

See Also

"Documenting a Code Generation Project"

rsimgetrtp

Purpose

Model's global parameter structure

Syntax

```
rsimgetrtp(model, option)
```

option is optional.

Arguments

model
: The model for which you are running the rapid simulations.

option (optional)
: The parameter-value pair `'AddTunableParamInfo' 'value'`, where *value* can be `'on'` or `'off'`. If you set the parameter to `'on'`, Real-Time Workshop extracts tunable parameter information from the specified model and returns it to *param_struct*.

Returns

A structure that contains the specified model's parameter structure.

Description

The `rsimgetrtp` function forces an update diagram action for the specified model and returns a structure that contains the following fields:

Field	Description
`modelChecksum`	A four-element vector that encodes the structure of the model. Real-Time Workshop uses the checksum to check whether the structure of the model has changed since the RSim executable was generated. If you delete or add a block, and then generate a new *`model`*`_P` vector, the new checksum no longer matches the original checksum. The RSim executable detects this incompatibility in parameter vectors and exits to avoid returning incorrect simulation results. If the model structure changes, you must regenerate the code for the model.
`parameters`	A structure that contains the model's global parameters.

The parameters substructure includes the following fields:

Field	Description
`dataTypeName`	The name of the parameter's data type, for example, `double`
`dataTypeID`	An internal data type identifier that Real-Time Workshop uses
`complex`	The value 0 if real and 1 if complex
`dtTransIdx`	Internal use only
`values`	A vector of parameter values

If you specify `'AddTunableParamInfo', 'on'`, Real-Time Workshop creates and then deletes *`model`*`.rtw` from your current working directory and includes a `map` substructure that has the following fields:

Field	Description
`Identifier`	Parameter name
`ValueIndicies`	A vector of indices to the parameter values
`Dimensions`	A vector indicating the parameter dimensions

To use the `AddTunableParamInfo` option, you must enable inline parameters.

Examples

Returns the parameter structure for model `rtwdemo_rsimtf` to `param_struct`.

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

param_struct =

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

See Also

"Creating a MAT-File That Includes a Model's Parameter Structure"

Purpose

Update files in model's 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 directory contains `myfile.c` and `myfile.cpp`, an instance of `myfile` would be updated to `myfile.c`.

Description

Using paths that already exist in a model's build information, the `updateFilePathsAndExtensions` function checks whether any 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

Examples

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

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

before =

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

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

ans =

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


ans =

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


ans =

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

See Also

`addIncludeFiles`, `addIncludePaths`, `addSourceFiles`, `addSourcePaths`, `updateFileSeparator`
"Programming a Post Code Generation Command"

updateFileSeparator

Purpose

Change file separator used in model's 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 all file path specifications.

Description

The `updateFileSeparator` function changes all instances of the current file separator (/ or \) in a model's 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-Compiling Code Generated on Windows". If the `GenerateMakefile` parameter is set, Real-Time Workshop overrides the default separator and updates the model's build information after evaluating the `PostCodeGenCommand` configuration parameter.

Examples

Update object `myModelBuildInfo` to apply the Windows file separator.

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

See Also

`addIncludeFiles`, `addIncludePaths`, `addSourceFiles`, `addSourcePaths`, `updateFilePathsAndExtensions`
"Programming a Post Code Generation Command", "Cross-Compiling Code Generated on Windows"

3

Simulink Block Support

The tables in this chapter summarize Real-Time Workshop and Real-Time Workshop Embedded Coder support for Simulink blocks. A table appears for each library. For each block, the second column indicates any support notes (SNs), which give information you will need when using the block for code generation.

All support notes appear at the end of this chapter in Support Notes on page 3-18. For more detail, enter the command `showblockdatatypetable` at the MATLAB command prompt or consult the block reference pages.

Additional Math and Discrete: Additional Discrete

Block	Support Notes
Fixed-Point State-Space	SN1
Transfer Fcn Direct Form II	SN1, SN2
Transfer Fcn Direct Form II Time Varying	SN1, SN2
Unit Delay Enabled	SN1, SN2
Unit Delay Enabled External IC	SN1, SN2
Unit Delay Enabled Resettable	SN1, SN2
Unit Delay Enabled Resettable External IC	SN1, SN2
Unit Delay External IC	SN1, SN2
Unit Delay Resettable	SN1, SN2
Unit Delay Resettable External IC	SN1, SN2
Unit Delay With Preview Enabled	SN1, SN2
Unit Delay With Preview Enabled Resettable	SN1, SN2
Unit Delay With Preview Enabled Resettable External RV	SN1, SN2
Unit Delay With Preview Resettable	SN1, SN2
Unit Delay With Preview Resettable External RV	SN1, SN2

Additional Math and Discrete: Increment/Decrement

Block	Support Notes
Decrement Real World	SN1
Decrement Stored Integer	SN1
Decrement Time To Zero	—
Decrement To Zero	SN1
Increment Real World	SN1
Increment Stored Integer	SN1

Continuous

Block	Support Notes
Derivative	SN3, SN4
Integrator	SN3, SN4
State-Space	SN3, SN4
Transfer Fcn	SN3, SN4
Transport Delay	SN3, SN4
Variable Time Delay	SN3, SN4
Variable Transport Delay	SN3, SN4
Zero-Pole	SN3, SN4

Discontinuities

Block	Support Notes
Backlash	SN2
Coulomb and Viscous Friction	SN1
Dead Zone	—
Dead Zone Dynamic	SN1
Hit Crossing	SN4
Quantizer	—
Rate Limiter	SN5
Rate Limiter Dynamic	SN1, SN5
Relay	—
Saturation	—
Saturation Dynamic	SN1
Wrap To Zero	SN1

Discrete

Block	Support Notes
Difference	SN1
Discrete Derivative	SN2, SN6
Discrete Filter	SN2
Discrete State-Space	SN2
Discrete Transfer Fcn	SN2
Discrete Zero-Pole	SN2
Discrete-Time Integrator	SN2, SN6
First-Order Hold	SN4
Integer Delay	SN2
Memory	—
Tapped Delay	SN2
Transfer Fcn First Order	SN1
Transfer Fcn Lead or Lag	SN1
Transfer Fcn Real Zero	SN1
Unit Delay	SN2
Weighted Moving Average	—
Zero-Order Hold	—

Logic and Bit Operations

Block	Support Notes
Bit Clear	—
Bit Set	—
Bitwise Operator	—
Combinatorial Logic	—
Compare to Constant	—
Compare to Zero	—
Detect Change	SN2
Detect Decrease	SN2
Detect Fall Negative	SN2
Detect Fall Nonpositive	SN2
Detect Increase	SN2
Detect Rise Nonnegative	SN2
Detect Rise Positive	SN2
Extract Bits	—
Interval Test	—
Interval Test Dynamic	—
Logical Operator	—
Relational Operator	—
Shift Arithmetic	—

Lookup Tables

Block	Support Notes
Cosine	SN1
Direct Lookup Table (n-D)	SN2
Interpolation Using Prelookup	—
Lookup Table	—
Lookup Table (2-D)	—
Lookup Table (n-D)	—
Lookup Table Dynamic	—
Prelookup	—
Sine	SN1

Math Operations

Block	Support Notes
Abs	—
Add	—
Algebraic Constraint	Not supported
Assignment	SN2
Bias	—
Complex to Magnitude-Angle	—
Complex to Real-Imag	—
Divide	SN2
Dot Product	—
Gain	—
Magnitude-Angle to Complex	—
Math Function (10^u)	—
Math Function (conj)	—
Math Function (exp)	—
Math Function (hermitian)	—
Math Function (hypot)	—
Math Function (log)	—
Math Function (log10)	—
Math Function (magnitude^2)	—
Math Function (mod)	—
Math Function (pow)	—
Math Function (reciprocal)	—
Math Function (rem)	—
Math Function (square)	—
Math Function (sqrt)	—

Math Operations (Continued)

Block	Support Notes
Math Function (transpose)	—
Matrix Concatenate	SN2
MinMax	—
MinMax Running Resettable	—
Permute Dimensions	SN2
Polynomial	—
Product	SN2
Product of Elements	SN2
Real-Imag to Complex	—
Reshape	—
Rounding Function	—
Sign	—
Sine Wave Function	SN6, SN9
Slider Gain	—
Squeeze	SN2
Subtract	—
Sum	—
Sum of Elements	—
Trigonometric Function	SN7
Unary Minus	—
Vector Concatenate	SN2
Weighted Sample Time Math	—

Model Verification

Block	Support Notes
Assertion	—
Check Discrete Gradient	—
Check Dynamic Gap	—
Check Dynamic Lower Bound	—
Check Dynamic Range	—
Check Dynamic Upper Bound	—
Check Input Resolution	—
Check Static Gap	—
Check Static Lower Bound	—
Check Static Range	—
Check Static Upper Bound	—

Ports & Subsystems

Block	Support Notes
Atomic Subsystem	—
CodeReuse Subsystem	—
Configurable Subsystem	—
Enabled Subsystem	—
Enabled and Triggered Subsystem	—
For Iterator Subsystem	—
Function-Call Generator	—
Function-Call Subsystem	—
If	—
If Action Subsystem	—
Model	—
Subsystem	—
Switch Case	—
Switch Case Action Subsystem	—
Triggered Subsystem	—
While Iterator Subsystem	—

Signal Attributes

Block	Support Notes
Bus to Vector	—
Data Type Conversion	—
Data Type Conversion Inherited	—
Data Type Duplicate	—
Data Type Propagation	—
Data Type Scaling Strip	—
IC	SN4
Probe	—
Rate Transition	SN2, SN5
Signal Conversion	—
Signal Specification	—
Weighted Sample Time	—
Width	—

Signal Routing

Block	Support Notes
Bus Assignment	—
Bus Creator	—
Bus Selector	—
Data Store Memory	—
Data Store Read	—
Data Store Write	—
Demux	—
Environment Controller	—
From	—
Goto	—
Goto Tag Visibility	—
Index Vector	—
Manual Switch	SN4
Merge	SN13
Multiport Switch	SN2
Mux	—
Selector	—
Switch	SN2

Sinks

Block	Support Notes
Display	SN8
Floating Scope	SN8
Outport (Out1)	—
Scope	SN8
Stop Simulation	SN14
Terminator	—
To File	SN4
To Workspace	SN8
XY Graph	SN8

Sources

Block	Support Notes
Band-Limited White Noise	SN5
Chirp Signal	SN4
Clock	SN4
Constant	—
Counter Free-Running	SN4
Counter Limited	SN1, SN4
Digital Clock	SN4
From File	SN8
From Workspace	SN8
Ground	—
Inport (In1)	—
Pulse Generator	SN5, SN9
Ramp	SN4
Random Number	—
Repeating Sequence	SN10
Repeating Sequence Interpolated	SN1, SN5
Repeating Sequence Stair	SN1
Signal Builder	SN4
Signal Generator	SN4
Sine Wave	SN6, SN9
Step	SN4
Uniform Random Number	—

User-Defined

Block	Support Notes
Embedded MATLAB Function	—
Fcn	—
Level-2 M-File S-Function	Not supported
MATLAB Fcn	SN11
S-Function	SN12
S-Function Builder	—

Support Notes

Symbol	Note
—	Real-Time Workshop supports the block and requires no special notes.
SN1	Real-Time Workshop does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more optimal code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.
SN2	Generated code relies on `memcpy` or `memset` (`string.h`) under certain conditions.
SN3	Consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support code generation. To start the Model Discretizer, click **Tools > Control Design**.
SN4	Not recommended for production code.
SN5	Cannot use inside a triggered subsystem hierarchy.
SN6	Depends on absolute time when used inside a triggered subsystem hierarchy.
SN7	The three functions — `asinh`, `acosh`, and `atanh` — are not supported by all compilers. If you use a compiler that does not support these functions, Real-Time Workshop issues a warning message for the block and the generated code fails to link.
SN8	Ignored for code generation.
SN9	Does not refer to absolute time when configured for sample-based operation. Depends on absolute time when in time-based operation.
SN10	Consider using the Repeating Sequence Stair or Repeating Sequence Interpolated block instead.
SN11	Consider using the Embedded MATLAB block instead.

Support Notes (Continued)

Symbol	Note
SN12	S-functions that call into MATLAB are not supported for code generation.
SN13	When more than one signal connected to a Merge block has a non-`Auto` storage class, all non-`Auto` signals connected to that block must *be identically labeled* and *have the same storage class*. When Merge blocks connect directly to one another, these rules apply to all signals connected to any of the Merge blocks in the group.
SN14	When a model includes a Stop Simulation block, generated code stops executing when the stop condition is true.

4

Blocks — By Category

Custom Code

Model Header	Specify custom header code
Model Source	Specify custom source code
System Derivatives	Specify custom system derivative code
System Disable	Specify custom system disable code
System Enable	Specify custom system enable code
System Initialize	Specify custom system initialization code
System Outputs	Specify custom system outputs code
System Start	Specify custom system startup code
System Terminate	Specify custom system termination code
System Update	Specify custom system update code

Interrupt Templates

Async Interrupt	Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that are to execute downstream subsystems or Task Sync blocks
Task Sync	Spawn VxWorks task to run code of downstream function-call subsystem or Stateflow chart

S-Function Target

RTW S-Function	Represent model or subsystem as generated S-function code

VxWorks

Async Interrupt	Generate Versa Module Eurocard (VME) interrupt service routines (ISRs) that are to execute downstream subsystems or Task Sync blocks
Protected RT	Handle transfer of data between blocks operating at different rates and ensure data integrity
Task Sync	Spawn VxWorks task to run code of downstream function-call subsystem or Stateflow chart
Unprotected RT	Handle transfer of data between blocks operating at different rates and ensure determinism

5

Blocks — Alphabetical List

Async Interrupt

Purpose

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

Library

Interrupt Templates, VxWorks

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. Real-Time Workshop 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 the proper rate transition code (see "Rate Transitions and Asynchronous Blocks" in the Real-Time Workshop documentation). Simulink task priority values are also required to ensure 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 Simulink 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 all 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 appropriate timer register on the target hardware. See “Using Timers in Asynchronous Tasks” and “Async Interrupt Block Implementation” in the Real-Time Workshop 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`, Real-Time Workshop 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 Real-Time Workshop to handle as a 32-bit integer of the specified resolution, Real-Time Workshop uses a second 32-bit integer to address overflows.

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

Enable simulation input

If checked, Simulink 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 ensure that those blocks do not contribute to the generated code. Alternatively, you can use the Environment Controller block, as explained in "Dual-Model Approach: Code Generation". However, if you use the Environment Controller block, be aware that the sample times of driving blocks contribute to the sample times supported in the generated code.

Async Interrupt

Inputs and Outputs

Input

A simulated interrupt source.

Output

Control signal for a

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

Assumptions and Limitations

- The block supports VME interrupts 1 through 7.
- The block requires a VxWorks Board Support Package (BSP) that supports the following VxWorks system calls:

```
sysIntEnable
sysIntDisable
intConnect
intLock
intUnlock
tickGet
```

Performance Considerations

Execution of large subsystems at interrupt level can have a significant impact on interrupt response time for interrupts of equal and lower priority in the system. As a general rule, it is best to keep ISRs as short as possible. Connect only function-call subsystems that contain a small number of blocks to an Async Interrupt block.

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

See Also

Task Sync
"Asynchronous Support" in the Real-Time Workshop documentation

Model Header

Purpose

Specify custom header code

Library

Custom Code

Description

The Model Header block adds user-specified custom code to the *model*.h file that Real-Time Workshop generates for the model that contains the block.

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 at the top of the model's generated header file.

Bottom of Model Header

Code to be added at the top of the model's generated header file.

Example

See "Example: Using a Custom Code Block".

See Also

Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
"Inserting Custom Code Into Generated Code" in the Real-Time Workshop documentation

Purpose

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 Real-Time Workshop generates for the model that contains the block.

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 at the top of the model's generated source file.

Bottom of Model Source

Code to be added at the top of the model's generated source file.

Example

See "Example: Using a Custom Code Block".

See Also

Model Header, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
"Inserting Custom Code Into Generated Code" in the Real-Time Workshop documentation

Protected RT

Purpose

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

Library

VxWorks

Description

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

Purpose

Represent model or subsystem as generated S-function code

Library

S-Function Target

Description

An instance of the RTW S-Function block represents code Real-Time Workshop generates from its S-function target for a model or subsystem. For example, you extract a subsystem from a model and build an RTW 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
- Protecting intellectual property — you need only provide the binary MEX-file object to users

For details on how to create an RTW S-Function block from a subsystem, see "Creating an S-Function Block from a Subsystem" in the Real-Time Workshop documentation.

Requirements

- The S-Function block must perform identically to the model or subsystem from which it was generated.
- Before creating the block, you must explicitly specify all 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" in the Real-Time Workshop documentation.
- You must set the solver parameters of the RTW S-function block to be the same as those of the original model or subsystem. This ensures that the generated S-function code will operate identically to the original subsystem (see Choice of Solver Type in the Real-Time Workshop documentation for an exception to this rule).

Parameters

Generated S-function name (model_sf)

The name of the generated S-function. Real-Time Workshop 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

"Creating an S-Function Block from a Subsystem" in the Real-Time Workshop documentation

Purpose

Specify custom system derivative code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update
“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

System Disable

Purpose

Specify custom system disable code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Derivatives, System Enable, System Initialize, System Outputs, System Start, System Terminate, System Update

“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

Purpose

Specify custom system enable code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Derivatives, System Disable, System Initialize, System Outputs, System Start, System Terminate, System Update
“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

System Initialize

Purpose

Specify custom system initialization code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Derivatives, System Disable, System Enable, System Outputs, System Start, System Terminate, System Update
“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

Purpose

Specify custom system outputs code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Start, System Terminate, System Update
“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

System Start

Purpose

Specify custom system startup code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Terminate, System Update
“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

Purpose

Specify custom system termination code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Update
“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

System Update

Purpose

Specify custom system update code

Library

Custom Code

Description

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

Note If you include this block in a submodel (model referenced by a Model block), Real-Time Workshop 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 “Example: Using a Custom Code Block”.

See Also

Model Header, Model Source, System Derivatives, System Disable, System Enable, System Initialize, System Outputs, System Start, System Terminate
“Inserting Custom Code Into Generated Code” in the Real-Time Workshop documentation

Purpose

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

Library

Interrupt Templates, VxWorks

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 Simulink 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 all 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 "Using Timers in Asynchronous Tasks" in the Real-Time Workshop 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`, Real-Time Workshop 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 Real-Time Workshop to handle as a 32-bit integer of the specified resolution, Real-Time Workshop uses a second 32-bit integer to address overflows.

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

Inputs and Outputs

Input

A call from an Async Interrupt block.

Output

A call to a function-call subsystem.

See Also

Async Interrupt

"Asynchronous Support" in the Real-Time Workshop documentation

Purpose

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

Library

VxWorks

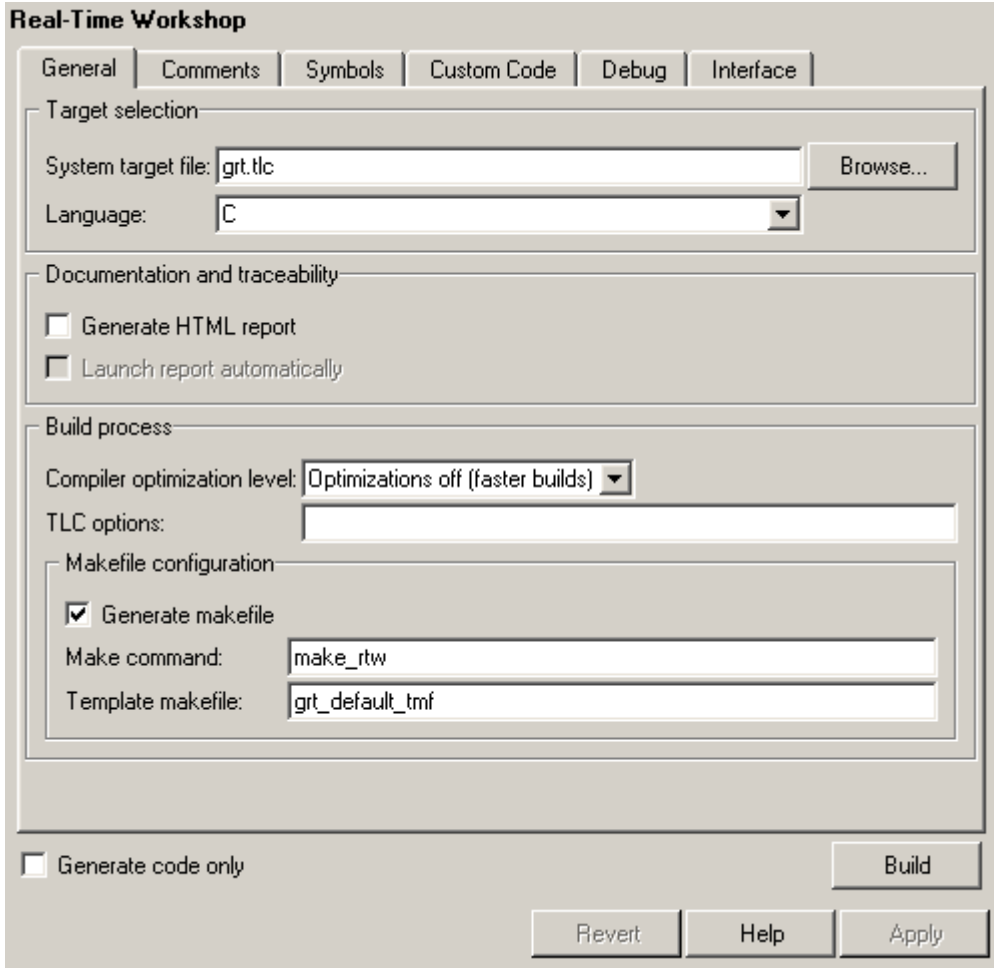
Description

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

Configuration Parameters

Real-Time Workshop Pane: General (p. 6-3)	General parameters for defining code generation for a model's active configuration set, including target selection, documentation, and build process
Real-Time Workshop Pane: Comments (p. 6-34)	Parameters for controlling the comments that you want to automatically generate and insert into the generated code
Real-Time Workshop Pane: Symbols (p. 6-52)	Parameters for selecting automatically generated naming rules for identifiers in code generation
Real-Time Workshop Pane: Custom Code (p. 6-84)	Parameters for creating a list of custom C code, directories, source files, and libraries to include in generated files
Real-Time Workshop Pane: Debug (p. 6-94)	Parameters for debugging the build process and selecting TLC process options
Real-Time Workshop Pane: Interface (p. 6-103)	Parameters for selecting the target software environment, output variable name modifier, and data exchange interface
Real-Time Workshop Pane: RSim Target (p. 6-156)	

Real-Time Workshop Pane: General



In this section...

In this section...

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.

See Also

Real-Time Workshop Pane

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.

Tips

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

Command-Line Information

Parameter: `SystemTargetFile`
Type: string
Value: any valid system target file
Default: `'grt.tlc'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	ERT based (requires Real-Time Workshop Embedded Coder license)

See Also

- Available Targets
- Generating Efficient Code with Optimized ERT Targets
- Auto-Configuring Models for Code Generation
- Creating and Using Host-Based Shared Libraries

Language

Specify C or C++ code generation.

Settings

Default: `C`

`C`
: Generates `.c` files and places the files in your build directory.

`C++`
: Generates `.cpp` files and places the files in your build directory.

Tip

You might need to configure Real-Time Workshop to use the appropriate compiler before you build a system.

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

Choosing and Configuring a Compiler

Generate HTML 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` directory within the build directory. In the report,

- There is a summary listing version and date information, and a link to open configuration settings used for generating the code, including TLC options and Simulink model settings.
- Global variable instances are hyperlinked to their definitions.
- Block header comments in source files are hyperlinked back to the model; clicking one of these causes the block that generated that section of code to be highlighted (this feature requires Real-Time Workshop Embedded Coder and the ERT target).

 Off

Does not generate a summary of files.

Dependency

This parameter enables

- **Launch report automatically**
- **Code-to-block highlighting**
- **Block-to-code highlighting**

.

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

Generate HTML Report

If you are licensed to use Real-Time Workshop Embedded Coder, see also Generating and Using an HTML Code Generation Report.

Launch report automatically

Specify whether to display HTML reports automatically.

Settings

Default: off

 On

Displays the HTML report automatically in a new browser window.

 Off

Does not display the HTML report, but the report is still available in the `html` directory.

Dependency

This parameter is enabled by **Generate HTML report**.

Command-Line Information

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

Recommended Settings

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

See Also

Generate HTML Report

If you are licensed to use Real-Time Workshop Embedded Coder, see also Generating and Using an HTML Code Generation Report.

Code-to-block highlighting

Include hyperlinks in a generated HTML report that link code to the corresponding blocks in the model diagram.

Settings

Default: off

 On

Includes hyperlinks in the generated HTML report that link code to corresponding blocks 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 option 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 is enabled by **Generate HTML report**.

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

Generating and Using an HTML Code Generation Report.

Block-to-code highlighting

Links blocks 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 block-to-code highlighting support in the generated HTML report. To highlight the generated code for a block in the HTML report, right-click the block and select **Real-Time Workshop > Highlight Code**.

 Off

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

Tip

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter is enabled by **Generate HTML report**.

Command-Line Information

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

Recommended Settings

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

See Also

Generating and Using an HTML Code Generation Report.

Configure

Use the **Configure** button to open the Block-to-code highlighting dialog box. This dialog box provides a way for you to specify a build directory containing previously-generated model code to highlight. Applying your build directory selection will attempt to load traceability information from the earlier build, for which **Block-to-code highlighting** must have been selected.

Dependency

This parameter only appears for ERT-based targets.

See Also

Generating and Using an HTML Code Generation Report.

Compiler optimization level

Provides flexible and generalized control over compiler optimizations for building generated code.

Settings

Default: `Optimizations off (faster builds)`

`Optimizations off (faster builds)`
: Customizes compilation during the Real-Time Workshop makefile build process to minimize compilation time.

`Optimizations on (faster runs)`
: Customizes compilation during the Real-Time Workshop makefile build process to minimize run time.

`Custom`
: Allows you to specify custom compiler optimization flags to be applied during the Real-Time Workshop 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 Real-Time Workshop make commands.
- If you specify compiler options for your Real-Time Workshop 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)`
Safety precaution	No impact

See Also

- Custom compiler optimization flags
- Controlling Compiler Optimization Level and Specifying Custom Optimization Settings

Custom compiler optimization flags

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

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.

See Also

- Compiler optimization level
- Controlling Compiler Optimization Level and Specifying Custom Optimization Settings

TLC options

Specify Target Language Compiler (TLC) options for code generation.

Settings

Default: `''`

You can enter TLC command-line options and arguments.

Tips

- Specifying TLC options does not add flags to the **Make command** field.
- The summary section of the generated HTML report lists the TLC options that you specify for the build in which you generate the report.

Command-Line Information

Parameter: `TLCOptions`
Type: string
Value: any valid TLC argument
Default: `''`

Recommended Settings

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

See Also

- TLC Options
- Command-Line Arguments

- Customizing the Target Build Process with the STF_make_rtw Hook File
- Understanding and Using the Build Process

Generate makefile

Specify generation of a makefile.

Settings

Default: on

 On

Generates a makefile for a model during the build process.

 Off

Suppresses the generation of a makefile. When you clear this parameter, you must set up any 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

- Customizing Post Code Generation Build Processing
- Customizing the Target Build Process with the STF_make_rtw Hook File
- Understanding and Using the Build Process

Make command

Specify a make command.

Settings

Default: `make_rtw`

The make command, a high-level M-file command, invoked when you start a build, controls the Real-Time Workshop build process.

- Each target has an associated make command, automatically supplied when you select a target file using the System Target File Browser.
- Some third-party targets supply a make command. See the vendor's documentation.
- You can specify arguments in the **Make command** field which pass into the makefile-based build process.

Tip

Most targets use the default command.

Dependency

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

Command-Line Information

Parameter: `MakeCommand`
Type: string
Value: any valid make command M-file
Default: `'make_rtw'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	`make_rtw`

See Also

- Template Makefiles and Make Options
- Customizing the Target Build Process with the STF_make_rtw Hook File
- Understanding and Using the Build Process

Template makefile

Specify a template makefile.

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, Real-Time Workshop attempts to find and execute an M-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: any 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.

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

Generate code only

Specify code generation versus an executable build.

Settings

Default: off

 On

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

 Off

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

Tip

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

Dependency

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

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Customizing 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'`

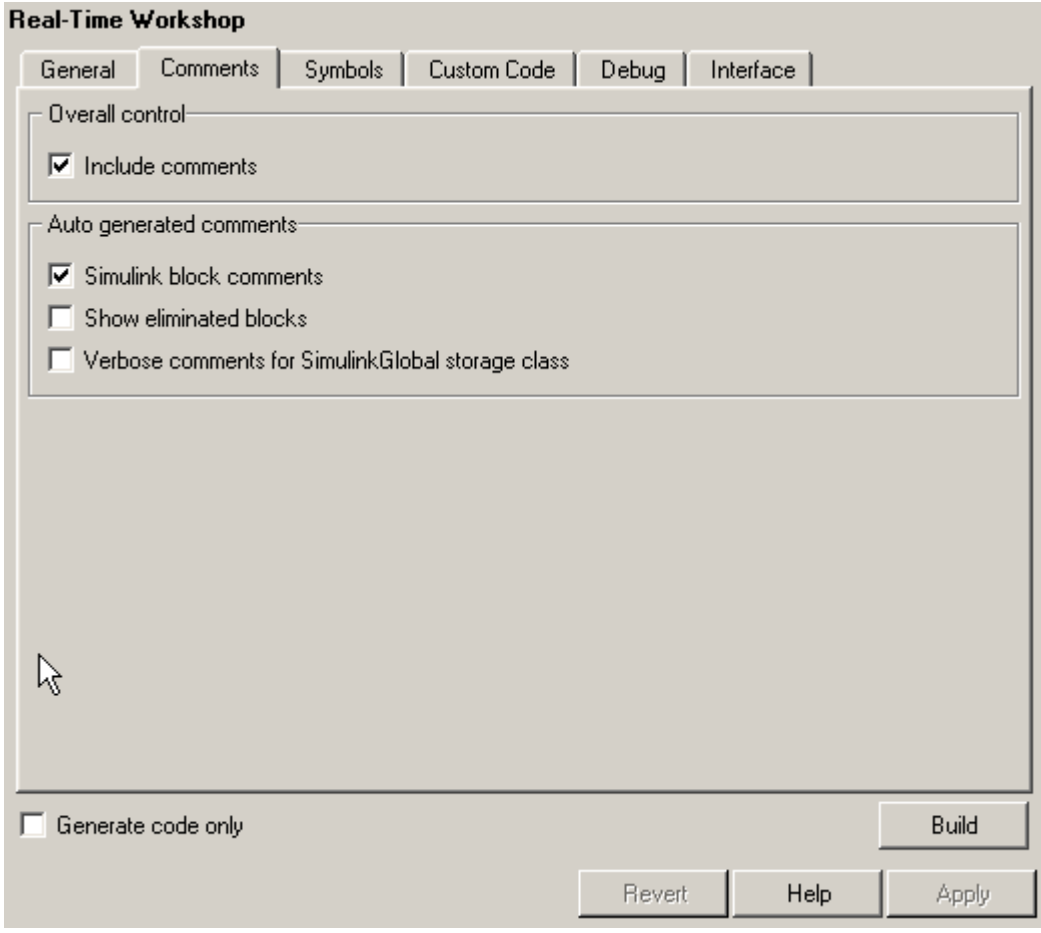
Recommended Settings

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

See Also

Initiating the Build Process

Real-Time Workshop Pane: Comments



In this section...

In this section...

Comments Tab Overview

Control the comments that Real-Time Workshop automatically generates and inserts into the generated code.

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.

Dependencies

This parameter enables:

- **Simulink block 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 comments

Specify whether to insert Simulink block comments.

Settings

Default: on

 On

Inserts automatically generated comments that describe a block's code. 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

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: `ShowEliminatedStatements`
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

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 comments**
- Text specified in the **Description** field of each Block Parameter 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.

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

Support for International (Non-US-ASCII) Characters

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.

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 that you include the comments in a function defined in an M-file or TLC file that you specify with **Custom comments function**.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Adding Custom Comments

Custom comments function

Specify a file that contains comments to be included in generated code for module packing tool (MPT) signal and parameter data objects

Settings

Default: `''`

Enter the name of the M-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 an object's property values.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter is enabled by **Custom comments (MPT objects only)**.

Command-Line Information

Parameter: `CustomCommentsFcn`
Type: string
Value: any valid file name
Default: `''`

Recommended Settings

Application	Setting
Debugging	Any valid file name
Traceability	Any valid file name
Efficiency	No impact
Safety precaution	No impact

See Also

Adding Custom Comments

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.

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

Support for International (Non-US-ASCII) Characters

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. Real-Time Workshop 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 has no 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.

Command-Line Information

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

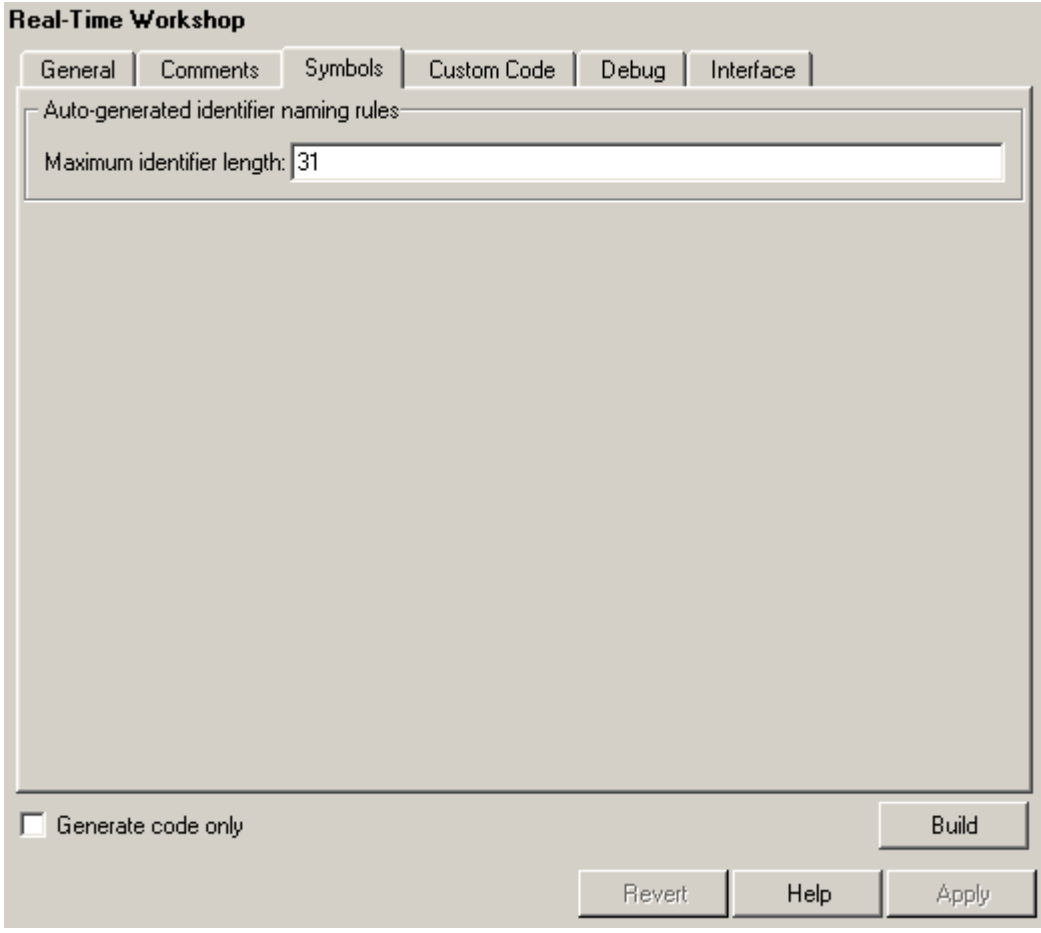
Recommended Settings

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

See Also

Including Requirements with Generated Code in the Simulink Verification and Validation documentation

Real-Time Workshop Pane: Symbols



In this section...

In this section...

Symbols Tab Overview

Select the automatically generated identifier naming rules.

See Also

Symbols Options

Global variables

Customize generated global variable identifiers.

Settings

Default: `$R$N$M`

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any 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 affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `CustomSymbolStrGlobalVar`
Type: string
Value: any valid combination of tokens
Default: `'$R$N$M'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combination of tokens
Efficiency	No impact
Safety precaution	`$R$N$M`

See Also

- Specifying Identifier Formats
- Name Mangling
- Model Referencing Considerations
- Identifier Format Control Parameters Limitations

Global types

Customize generated global type identifiers.

Settings

Default: `$N$R$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 any 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 affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `CustomSymbolStrType`
Type: string
Value: any valid combination of tokens
Default: `'$N$R$M'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combination of tokens
Efficiency	No impact
Safety precaution	`$N$R$M`

See Also

- Specifying Identifier Formats
- Name Mangling
- Model Referencing Considerations

- Identifier Format Control Parameters Limitations

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
`$A`	Insert data type acronym (for example, `i32` for long integers) into signal and work vector identifiers.
`$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 Simulink.
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$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 affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `CustomSymbolStrField`
Type: string
Value: any valid combination of tokens
Default: `'$N$M'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combination of tokens
Efficiency	No impact
Safety precaution	`$N$M`

See Also

- Specifying Identifier Formats
- Name Mangling
- Identifier Format Control Parameters Limitations

Subsystem methods

Customize generated global type identifiers.

Settings

Default: `$R$N$M$F`

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
`$F`	Insert method name (for example, `_Update` for update method). Empty for Stateflow functions.
`$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 Simulink. Empty for Stateflow functions.
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any 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 affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `CustomSymbolStrFcn`
Type: string
Value: any valid combination of tokens
Default: `'$R$N$M$F'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combination of tokens
Efficiency	No impact
Safety precaution	`$R$N$M$F`

See Also

- Specifying Identifier Formats
- Name Mangling
- Model Referencing Considerations
- Identifier Format Control Parameters Limitations

Local temporary variables

Customize generated local temporary variable identifiers.

Settings

Default: `$N$M`

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$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 any 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 affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `CustomSymbolStrTmpVar`
Type: string
Value: any valid combination of tokens
Default: `'$N$M'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combination of tokens
Efficiency	No impact
Safety precaution	`$N$M`

See Also

- Specifying Identifier Formats
- Name Mangling
- Model Referencing Considerations
- Identifier Format Control Parameters Limitations

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
`$A`	Insert data type acronym (for example, `i32` for long integers) into signal and work vector identifiers.
`$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 affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `CustomSymbolStrBlkIO`
Type: string
Value: any valid combination of tokens
Default: `'rtb_$N$M'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combination of tokens
Efficiency	No impact
Safety precaution	`rtb_$N$M`

See Also

- Specifying Identifier Formats
- Name Mangling
- Identifier Format Control Parameters Limitations

Constant macros

Customize generated constant macro identifiers.

Settings

Default: `$R$N$M`

Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include a combination of the following format tokens.

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any 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 affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `CustomSymbolStrMacro`
Type: string
Value: any valid combination of tokens
Default: `'$R$N$M'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combination of tokens
Efficiency	No impact
Safety precaution	`$R$N$M`

See Also

- Specifying Identifier Formats
- Name Mangling
- Model Referencing Considerations
- Identifier Format Control Parameters Limitations

Minimum mangle length

Increase the minimum number of characters used for generating name mangling strings that help avoid name collisions.

Settings

Default: 1

Specify an integer value that indicates the minimum number of characters the code generator is to use when generating a name mangling string. As necessary, 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 and safe; 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 only appears for ERT-based targets.

Command-Line Information

Parameter: `MangleLength`
Type: integer
Value: any valid value
Default: `1`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	1
Efficiency	No impact
Safety precaution	4

See Also

- Name Mangling
- Traceability
- Minimizing Name Mangling

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 the name mangling string (if any). 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: any valid value
Default: `31`

Recommended Settings

Application	Setting
Debugging	Any valid value
Traceability	>30
Efficiency	No impact
Safety precaution	>30

See Also

Generating Code for Model Referencing

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 only appears for ERT-based targets.
- This parameter is enabled by **Inline parameters** .

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`
: Makes no change to 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 all uppercase characters when creating identifiers for signal names in the generated code.

`Force lower case`
: Uses all lowercase characters when creating identifiers for signal names in the generated code.

`Custom M-function`
: Uses the M-file function specified with the **M-function** parameter to create identifiers for signal names in the generated code.

Dependencies

- This parameter only appears for ERT-based targets.
- 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: `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

- Applying Naming Rules to Identifiers Globally
- M-File Programming

M-function

Specify rule for naming identifiers in generated code.

Settings

Default: `''`

Enter the name of an M-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 an M-file function include:

- Remove underscore characters from signal names.
- Add an underscore before uppercase characters in parameter names.
- Make all identifiers uppercase in generated code.

Tip

M-file must be in the MATLAB path.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter is enabled by **Signal naming**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `DefineNamingFcn`
Type: string
Value: any M-file
Default: `''`

Recommended Settings

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

See Also

- Applying Naming Rules to Identifiers Globally
- M-File Programming

Parameter naming

Specify rule for naming parameters in generated code.

Settings

Default: `None`

`None`
: Makes no change to 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 all uppercase characters when creating identifiers for parameter names in the generated code.

`Force lower case`
: Uses all lowercase characters when creating identifiers for parameter names in the generated code.

`Custom M-function`
: Uses the M-file function specified with the **M-function** parameter to create identifiers for parameter names in the generated code.

Dependencies

- This parameter only appears for ERT-based targets.
- 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

- Applying Naming Rules to Identifiers Globally
- M-File Programming

#define naming

Specify rule for naming `#define` parameters (defined with storage class `Define (Custom)`) in generated code.

Settings

Default: `None`

`None`
: Makes no change to `#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 all uppercase characters when creating identifiers for `#define` parameter names in the generated code.

`Force lower case`
: Uses all lowercase characters when creating identifiers for `#define` parameter names in the generated code.

`Custom M-function`
: Uses the M-file function specified with the **M-function** parameter to create identifiers for `#define` parameter names in the generated code.

Dependencies

- This parameter only appears for ERT-based targets.
- 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'`

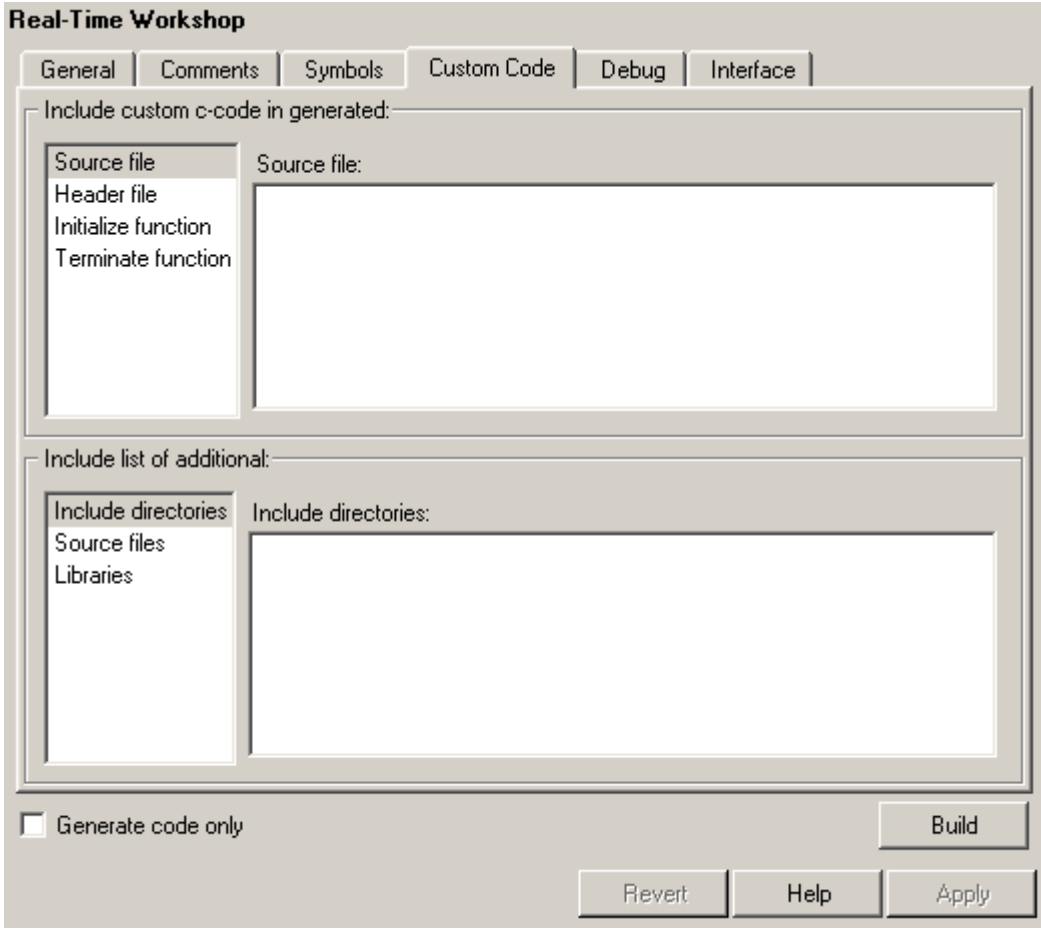
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Force upper case`
Efficiency	No impact
Safety precaution	No impact

See Also

- Applying Naming Rules to Identifiers Globally
- M-File Programming

Real-Time Workshop Pane: Custom Code



In this section...

In this section...

Custom Code Tab Overview

Create a list of custom C code, directories, source files, and libraries to include in generated files.

Configuration

1. Select the type of information to include from the list on the left side of the pane.
2. Enter a string to identify the specific code, directory, source file, or library.
3. Click **Apply**.

See Also

Configuring Custom Code

Source file

Specify a source file of code to appear at the top of generated files.

Settings

Default:`''`

Real-Time Workshop places code near the top of the generated *model*`.c` or *model*`.cpp` file, outside of any function.

Command-Line Information

Parameter: `CustomSource`
Type: string
Value: any source file name
Default: `''`

Recommended Settings

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

Header file

Specify a header file to include near top of generated file.

Settings

Default:`''`

Real-Time Workshop places header file code near the top of the generated *model*`.h` header file.

Command-Line Information

Parameter: `CustomHeaderCode`
Type: string
Value: any header file name
Default: `''`

Recommended Settings

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

Initialize function

Specify code appearing in an initialize function.

Settings

Default: `''`

Real-Time Workshop places code inside the model's initialize function in the *model*`.c` or *model*`.cpp` file.

Command-Line Information

Parameter: `CustomInitializer`
Type: string
Value: any code
Default: `''`

Recommended Settings

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

Terminate function

Specify code appearing in a 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 **Real-Time Workshop** pane, **Interface** tab.

Command-Line Information

Parameter: `CustomTerminator`
Type: string
Value: any code
Default: `''`

Recommended Settings

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

Include directories

Specify a list of include directories to add to the include path.

Settings

Default:`''`

Enter a space-separated list of include directories to add to the include path when compiling the generated code.

- Specify absolute or relative paths to the directories.
- Relative paths must be relative to the directory containing your model files, not relative to the build directory.
- The order in which you specify the directories is the order in which they are searched for source and include files.

Command-Line Information

Parameter: `CustomInclude`
Type: string
Value: any directory file name
Default: `''`

Recommended Settings

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

Source files

Specify the list of 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.

Tip

The file name is sufficient if the file is in the current MATLAB directory or in one of the include directories.

Command-Line Information

Parameter: `CustomSourceCode`
Type: string
Value: any source file name
Default: `''`

Recommended Settings

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

Libraries

Specify a list of additional libraries to link with the generated code.

Settings

Default: `''`

Enter a space-separated list of additional libraries to link with the generated code. Specify the libraries with a full path or just a file name when located in the current MATLAB directory or is listed as one of the include directories.

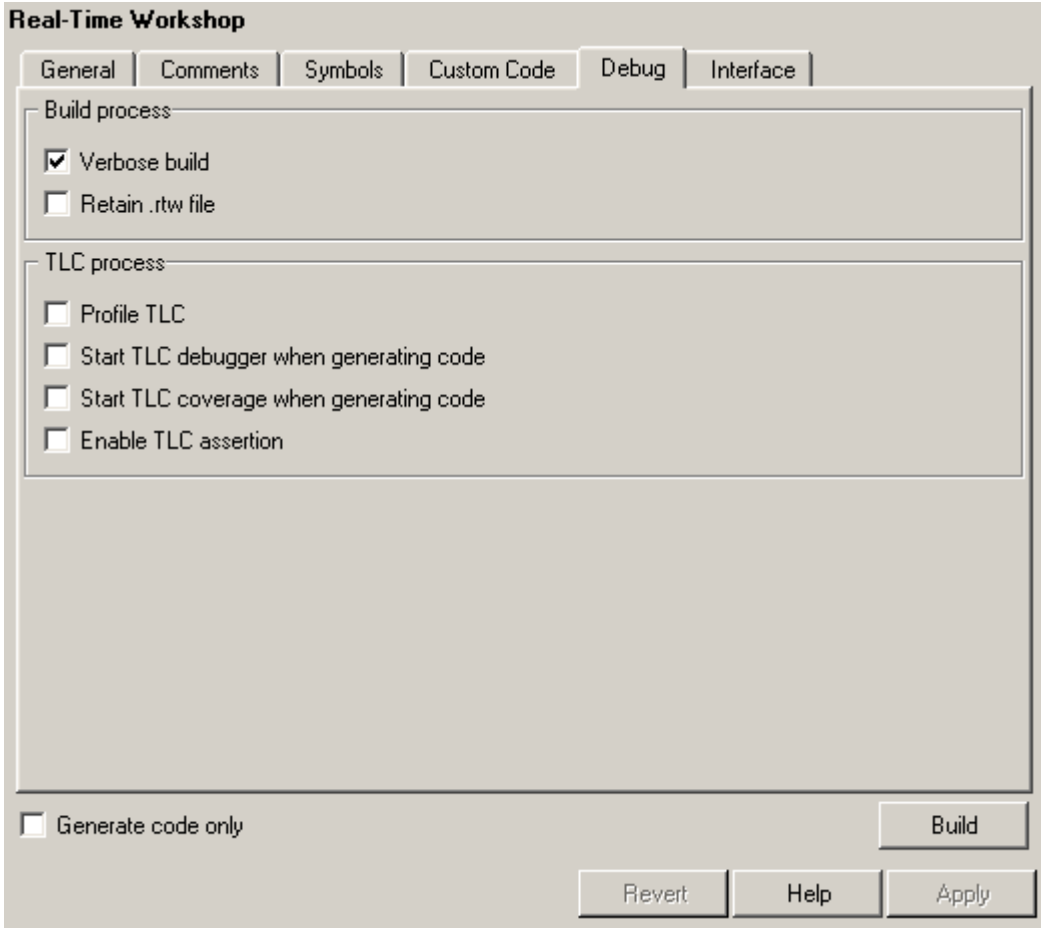
Command-Line Information

Parameter: `CustomLibrary`
Type: string
Value: any library file name
Default: `''`

Recommended Settings

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

Real-Time Workshop Pane: Debug



In this section...

In this section...

Debug Tab Overview

Select build process and Target Language Compiler (TLC) process options.

See Also

"Troubleshooting the Build Process"

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

Retain .rtw file

Specify *model*.rtw file retention.

Settings

Default: off

 On

Retains the *model*.rtw file in the current build directory. 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 directory 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

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

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

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

Enable TLC assertion

Produce the TLC stack trace

Settings

Default: off

 On

Real-Time Workshop halts building if any user-supplied TLC file contains an `%assert` directive that evaluates to FALSE.

 Off

Real-Time Workshop ignores TLC assertion code.

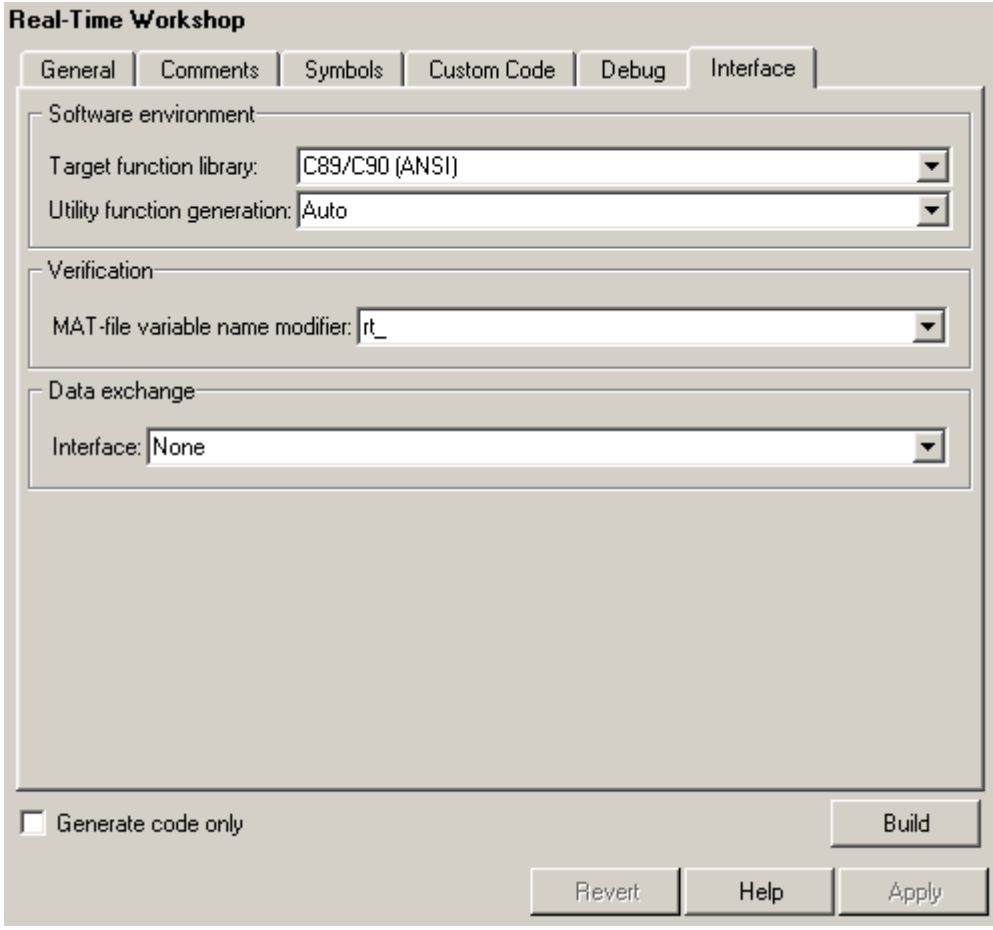
Command-Line Information

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

Recommended Settings

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

Real-Time Workshop Pane: Interface



In this section...

In this section...

Interface Tab Overview

Select the target software environment, output variable name modifier, and data exchange interface.

See Also

Configuring Model Interfaces

Target function library

Specify a target-specific 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.

`GNU99 (GNU)`
: Generates calls to the GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

Note Additional **Target function library** values may be listed if you have created and registered target function libraries with Real-Time Workshop Embedded Coder, or if you have licensed any Link or Target products. For more information on the **Target function library** values for Link or Target products, see your Link or Target product documentation.

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: `GenFloatMathFcnCalls`
Type: string
Value: `'ANSI_C' | 'C99 (ISO)' | 'GNU99 (GNU)'`
Default: `'ANSI_C'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Any valid library
Safety precaution	No impact

See Also

Configuring Model Interfaces

Utility function generation

Specify the location for generating utility functions.

Settings

Default: `Auto`

`Auto`
: Operates as follows:

- When the model contains Model blocks, place utilities within the `slprj/target/_sharedutils` directory.
- When the model does not contain Model blocks, place utilities in the build directory (generally, in *`model`*`.c` or *`model`*`.cpp`).

`Shared location`
: Directs code for utilities to be placed within the `slprj` directory in your working directory.

Command-Line Information

Parameter: `UtilityFuncGeneration`
Type: string
Value: `'Auto' | 'Shared location'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	`Shared location`
Traceability	`Shared location`
Efficiency	`Shared location`
Safety precaution	No impact

See Also

Configuring Model Interfaces

Support: floating-point numbers

Specify whether to generate floating-point data and operations.

Settings

Default: on

 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 parameter only appears for ERT-based targets.
- Selecting this parameter enables **Support: non-finite numbers** and clearing this parameter disables **Support: non-finite numbers**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	Off (for integer 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.
- 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

Timing Services

Support: non-finite numbers

Specify whether to generate nonfinite data and operations.

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.

Dependencies

- This parameter only appears 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

Application	Setting
Efficiency	Off
Safety precaution	Off

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 parameter only appears for ERT-based targets.
- This parameter must be on if your model includes blocks that require absolute or elapsed time values.
- This parameter must be off when generating an S-function wrapper for an ERT target; the code generator does not support continuous time for this target scenario.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	Off
Safety precaution	Off

See Also

- Support for Continuous Time Blocks, Continuous Solvers, and Stop Time
- Automatic S-Function Wrapper Generation

Support: complex numbers

Specify whether to generate complex data and operations.

Settings

Default: on

 On

Generates complex numbers and related operations.

 Off

Does not generate complex data and related operations. If you clear this option, an error occurs if the code generator encounters complex data or expressions. The error message reports offending blocks and parameters.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter 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: 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 you do not select this option and the model includes a C MEX S-function that does not have a corresponding TLC implementation (for inlining), 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.

Dependencies

- This parameter only appears for ERT-based targets.
- Selecting this parameter also selects **Support: floating-point numbers** and **Support: non-finite numbers**. If you subsequently clear **Support: floating-point numbers** or **Support: non-finite numbers**, a warning is displayed during code generation.

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

Automatic S-Function Wrapper Generation

GRT compatible call interface

Specify whether to generate model function calls compatible with the main program module of the GRT target.

Settings

Default: off

 On

Generates model function calls that are compatible with the main program module of the GRT target (`grt_main.c` or `grt_main.cpp`).

This option provides a quick way to use ERT target features with a GRT-based custom target that has a main program module based on `grt_main.c` or `grt_main.cpp`.

 Off

Disables the GRT compatible call interface.

Tips

The following are unsupported:

- Data type replacement
- Nonvirtual subsystem option **Function with separate data**

Dependencies

- This parameter only appears for ERT-based targets.
- Selecting this parameter also selects the required option **Support: floating-point numbers**. If you subsequently clear **Support: floating-point numbers**, an error is displayed during code generation.
- Selecting this parameter disables the incompatible option **Single output/update function**. Clearing this parameter enables **Single output/update function**.

Command-Line Information

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

Recommended Settings

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

See Also

Support for Continuous Time Blocks, Continuous Solvers, and Stop Time

Single output/update function

Specify whether to generate the *model*_step function.

Settings

Default: on

 On

Generates the *model*_step function for a model. This function contains the output and update function code for all 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 for model blocks in a single function. Generates the code in *model*.output and *model*.update.

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 Feedthrough" for details.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter and **GRT compatible call interface** are mutually incompatible and cannot both be selected through the GUI. Selecting **GRT compatible call interface** disables this parameter and clearing **GRT compatible call interface** enables this parameter.
- When you use this parameter, you must clear the option **Minimize algebraic loop occurrences** on the **Model Referencing** pane.

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

Terminate function required

Specify whether to generate the *model*_terminate function.

Settings

Default: on

 On

Generates a *model*_terminate function. This function contains all 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 must be the same for top-level and referenced models.

Command-Line Information

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

Recommended Settings

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

See Also

`model_terminate`

Generate reusable code

Specify whether to generate reusable, reentrant code.

Settings

Default: off

 On

Generates reusable, multi-instance code that is reentrant. The code generator passes model data structures (root-level inputs and outputs, block states, parameters, and external outputs) in, by reference, as arguments to *model*_step and other the model entry point functions. The data structures are also exported with *model*.h. For efficiency, the code generator passes in only data structures that are used. Therefore, when you select this option, the argument lists generated for the entry point functions vary according to model requirements.

 Off

Does not generate reusable code. Model data structures are statically allocated and accessed by model entry point functions directly in the model code.

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. If this option is selected, you must examine the generated code to determine the calling interface required for these functions.
- When this option is selected, the code generator generates a pointer to the real-time model object (*model*_M).
- In some cases, when this option is selected, the code generator might generate code that compiles but is not reentrant. For example, if any signal, DWork structure, or parameter data has a storage class other than Auto, global data structures are generated.

Dependencies

- This parameter only appears for ERT-based targets.

- This parameter enables **Reusable code error diagnostic** and **Pass root-level I/O as**.
- You must clear this option if you are using:
  - The static `ert_main.c` module, rather than generating a main program
  - 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 used by multiple instances has different sample times, data types, complexity, frame status, or dimension across the instances
    - Has output marked as a global signal
    - For each instance contains identical blocks with different names or parameter settings
- This parameter has no effect on code generated for function-call subsystems.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	On (for single instance)
Safety precaution	No impact

See Also

- Model Entry Points
- Nonvirtual Subsystem Code Generation
- Code Reuse Limitations
- Determining Why Subsystem Code Is Not Reused
- Writing S-Functions That Support Code Reuse
- Static Main Program Module
- Controlling model_step Function Prototypes
- Nonvirtual Subsystem Modular Function Code Generation
- Exporting Function-Call Subsystems
- `model_step`

Reusable code error diagnostic

Select the severity level for diagnostics displayed when a model violates requirements for generating reusable 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, Real-Time Workshop Embedded Coder might

- Generate code that compiles but is not reentrant. For example, if signal, DWork structure, or parameter data has a storage class other than `Auto`, global data structures are generated.
- Be unable to generate valid and compilable code. For example, if the model contains an S-function that is not code-reuse compliant or a subsystem triggered by a wide function-call trigger, the coder generates invalid code, displays an error message, and terminates the build.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter is enabled by **Generate reusable code**.

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

- Model Entry Points
- Nonvirtual Subsystem Code Generation
- Code Reuse Limitations
- Determining Why Subsystem Code Is Not Reused
- Nonvirtual Subsystem Modular Function Code Generation

Pass root-level I/O as

Control how root-level model input and output are passed to the *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 all 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.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter is enabled by **Generate reusable code**.

Command-Line Information

Parameter: `RootIOFormat`
Type: string
Value: `'Individual arguments' | 'Structure reference'`
Default: `'Individual arguments'`

Recommended Settings

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

See Also

- Model Entry Points
- Nonvirtual Subsystem Code Generation
- Nonvirtual Subsystem Modular Function Code Generation
- `model_step`

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.

Selecting this option can cause the code generator to completely 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 or set it with error message data.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter is cleared if you select the incompatible option **MAT-file logging**. If you subsequently select this parameter, an error is displayed during code generation.
- Selecting this parameter clears **Support: continuous time**.
- Setting of this parameter for multiple integrated models must match to avoid unexpected application behavior. For example, if you select the option for one model but not in another, an error status might not get registered by the integrated application.

Command-Line Information

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

Recommended Settings

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

See Also

rtModel Accessor Macros

Configure Functions

Use the **Configure Functions** button to open the Model Step Functions dialog box. This dialog box provides a way for you to specify whether the code generator is to use a default *model*_step function prototype or a model-specific C prototype. Based on your selection, you can preview and modify the function prototype.

Dependency

This parameter only appears for ERT-based targets.

See Also

- Controlling model_step Function Prototypes
- `model_step`
- Model Step Functions Dialog Box

Create Simulink (S-Function) block

Specify whether to generate an S-function block.

Settings

Default: off

 On

Generates an S-function block to represent the model or subsystem. The coder generates an inlined C or C++ MEX S-function wrapper that calls existing hand-written code or code previously generated by Real-Time Workshop from within Simulink. S-function wrappers provide a standard interface between Simulink and externally written code, allowing you to integrate your code into a model with minimal modification.

When this option is selected, Real-Time Workshop:

1. Generates the S-function wrapper file *`model`*`_sf.c` (or `.cpp`) and places it in the build directory.
2. Builds the MEX-file *`model`*`_sf.`*`mexext`* and places it in your working directory.
3. Creates and opens an untitled model containing the generated S-Function block.

 Off

Does not generate an S-function block.

Dependency

This parameter only appears for ERT-based targets.

Command-Line Information

Parameter: `GenerateErtSFunction`
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

- Automatic S-Function Wrapper Generation
- Techniques for Exporting Function-Call Subsystems
- Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes

Enable portable word sizes

Specify whether to allow portability across host and target processors that support different word sizes.

Settings

Default: off

 On

Generates conditional processing macros that support compilation of generated code on a processor that supports a different word size than the target processor on which production code is intended to run (for example, a 32-bit host and a 16-bit target. This allows you to use the same generated code for both software-in-the-loop (SIL) testing on the host platform and production deployment on the target platform.

 Off

Does not generate portable code.

Dependencies

- This parameter only appears for ERT-based targets.
- When you use this parameter, you should:
  - Select **Create Simulink (S-Function) block**
  - Set **Emulation hardware** on the **Hardware Implementation** pane to `None`

Command-Line Information

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

Recommended Settings

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

See Also

- Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes
- Tips for Optimizing the Generated 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 code generator saves system states, output, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) *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 needed 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

- This parameter only appears for ERT-based targets and the Tornado target.
- Selecting this parameter also selects the required options **Support: floating-point numbers**, **Support: non-finite numbers**, and **Terminate function required**. If you subsequently clear **Support: floating-point numbers**, **Support: non-finite numbers**, or **Terminate function required**, an error is displayed during code generation.
- Selecting this parameter 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 parameter enables **MAT-file variable name modifier**.
- Clear this option if you are using exported function calls.

Command-Line Information

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

Recommended Settings

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

See Also

Using 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

When an ERT target is selected, 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

Data Logging

Interface

Specify the data exchange interface (API) to include.

Settings

Default: `None`

`None`
: Does not include an API in the generated code.

`C-API`
: Uses the `C-API` data interface.

`External mode`
: Uses an external data interface.

`ASAP2`
: Uses the `ASAP2` data interface.

Dependencies

Selecting **C-API** enables the following parameters:

- **Signals in C API**
- **Parameters in C API**

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`, `ExtMode`, `GenerateASAP2`	`'off'`
C API	`RTWCAPIParams | RTWCAPISignals`	`'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

- C-API for Interfacing with Signals and Parameters
- External Mode
- Using External Mode with the ERT Target

Signals in C API

Generate a C API signal structure.

Settings

Default: on

 On

Generates C API for 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

C-API for Interfacing with Signals and Parameters

Parameters in C API

Generate C API parameter tuning structures.

Settings

Default: on

 On

Generates C API for global block and model 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

C-API for Interfacing with Signals and Parameters

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`.

`serial_win32`
: Applies a serial transport mechanism. The MEX-file name is `ext_serial_win32_comm`.

Tips

- The MEX-file is specified in `extmode-transports.m`.
- The MEX-file cannot be edited.

Dependency

This parameter is enabled by selecting `External mode` in the **Interface** parameter.

Command-Line Information

Parameter: `ExtModeTransport`
Type: integer
Value: `0 | 1`
Default: `0`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

- Creating an External Mode Communication Channel
- 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`)

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 target, this parameter is enabled by **Data Exchange > Interface > External mode** or by **External Mode**.

Command-Line Information

Parameter: `ExtModeMexArgs`
Type: string
Value: any valid arguments
Default: `""`

Recommended Settings

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

See Also

- Target Interfacing
- Client/Server Implementations

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 target, this parameter is enabled by **Data Exchange > Interface > External mode** or by **External Mode**.
- 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

External Mode Interface Options

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: any valid value
Default: `1000000`

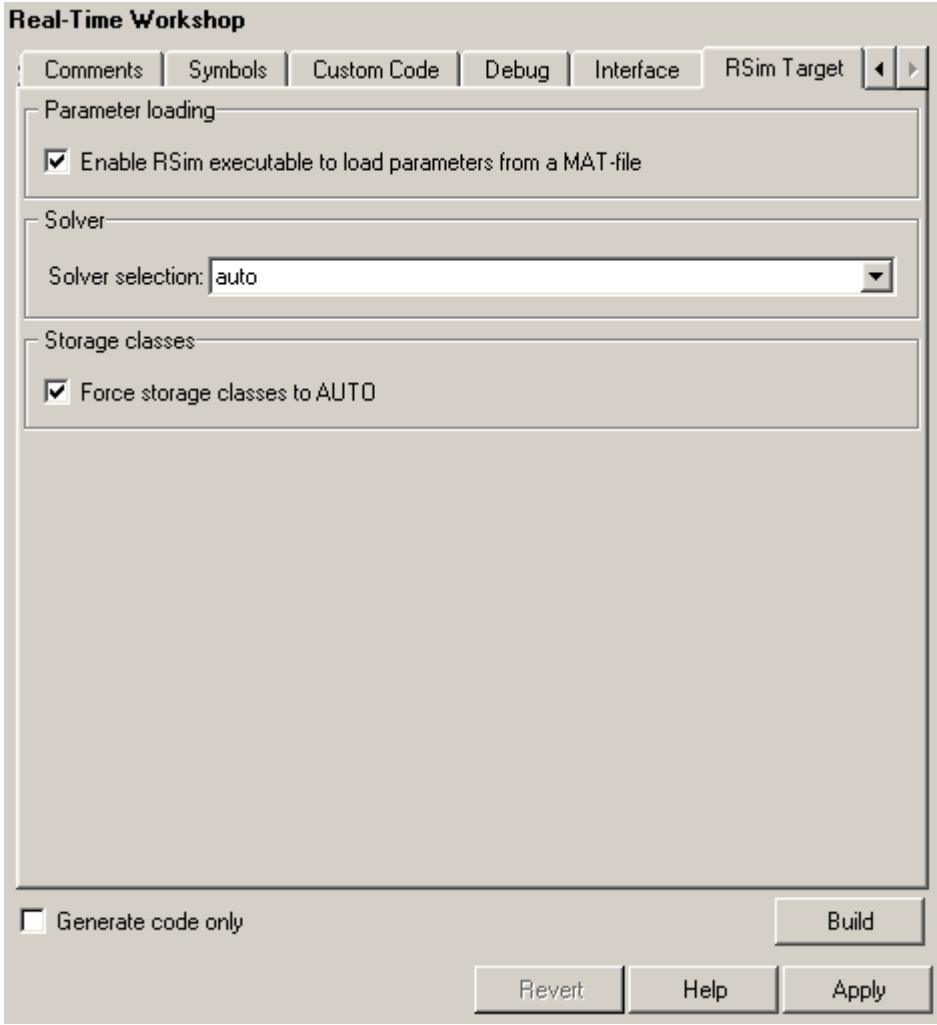
Recommended Settings

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

See Also

External Mode Interface Options

Real-Time Workshop Pane: RSim Target



In this section...

In this section...

RSim Target Tab Overview

Set configuration parameters for rapid simulation.

Configuration

This tab appears only if you specify the `rsim.tlc` system target file.

See Also

- Configuring and Building a Model for Rapid Simulation
- Running Rapid Simulations

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

Creating a MAT-File That Includes a Model's 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 Real-Time Workshop built-in solver.

`Use Simulink solver module`
: Instructs the target to use the variable-step solver that you specify on the Solver pane.

`Use Real-Time Workshop fixed-step solvers`
: Instructs the target to use the fixed-step solver that you specify on the Solver pane.

Tip

A Simulink license is checked out at run time if the executable includes the Simulink solver module.

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

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Licensing Protocols for Simulink Solvers in RSim Executables

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 Simulink to determine all 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 when it is necessary 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'`

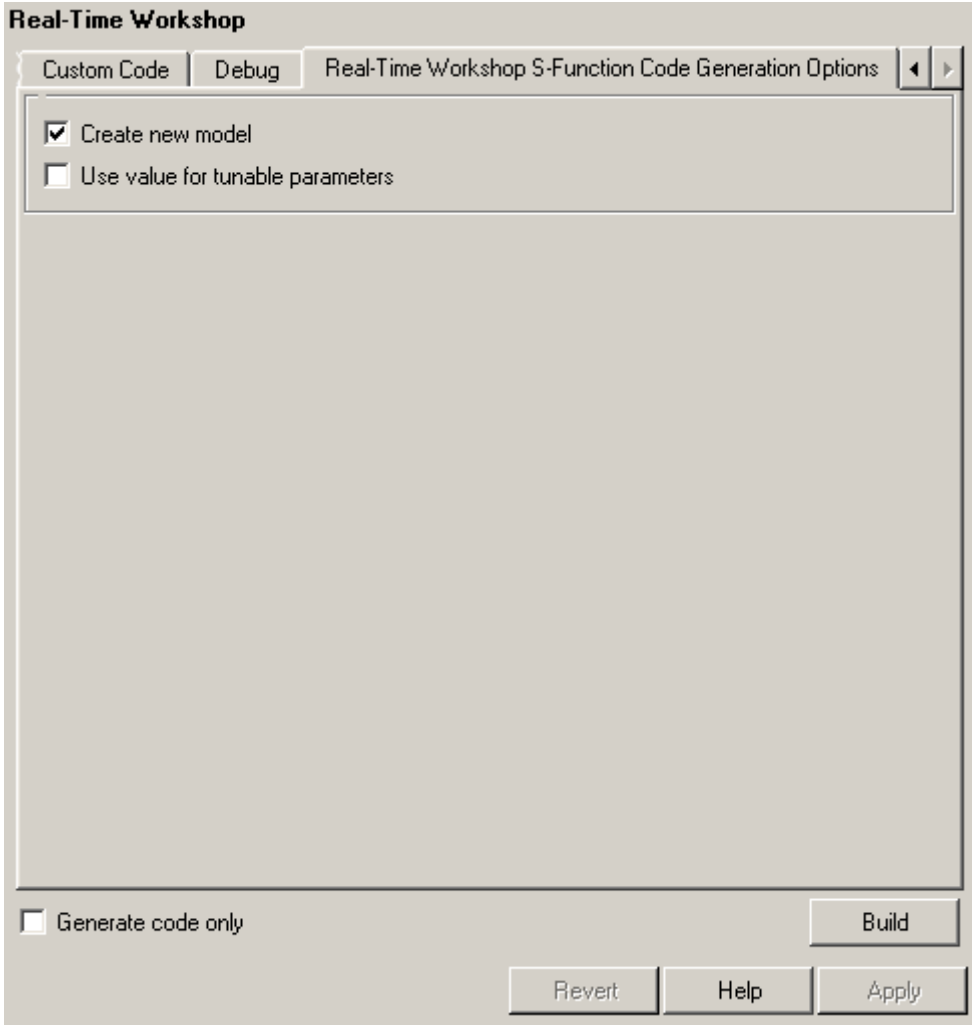
Recommended Settings

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

See Also

Licensing Protocols for Simulink Solvers in RSim Executables

Real-Time Workshop Pane: Real-Time Workshop S-Function Code Generation Options



In this section...

Real-Time Workshop S-Function Code Generation Options Tab Overview

Control the code generated by Real-Time Workshop for the S-function target (`rtwsfcn.tlc`).

Configuration

This tab appears only if you specify the S-function target (`rtwsfcn.tlc`) **System target file**. .

See Also

The S-Function Target

Create new model

Create a new model containing the generated Real-Time Workshop S-function block.

Settings

Default: on

 On

Creates a new model, separate from the current model, containing the generated Real-Time Workshop 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

The S-Function Target

Use value for tunable parameters

Use the variable value instead of the variable name in generated block mask edit fields for tunable parameters.

Settings

Default: off

 On

Uses variable values for tunable parameters instead of the variable name in the generated block mask edit fields.

 Off

Uses variable names for tunable parameters in the generated block mask edit fields.

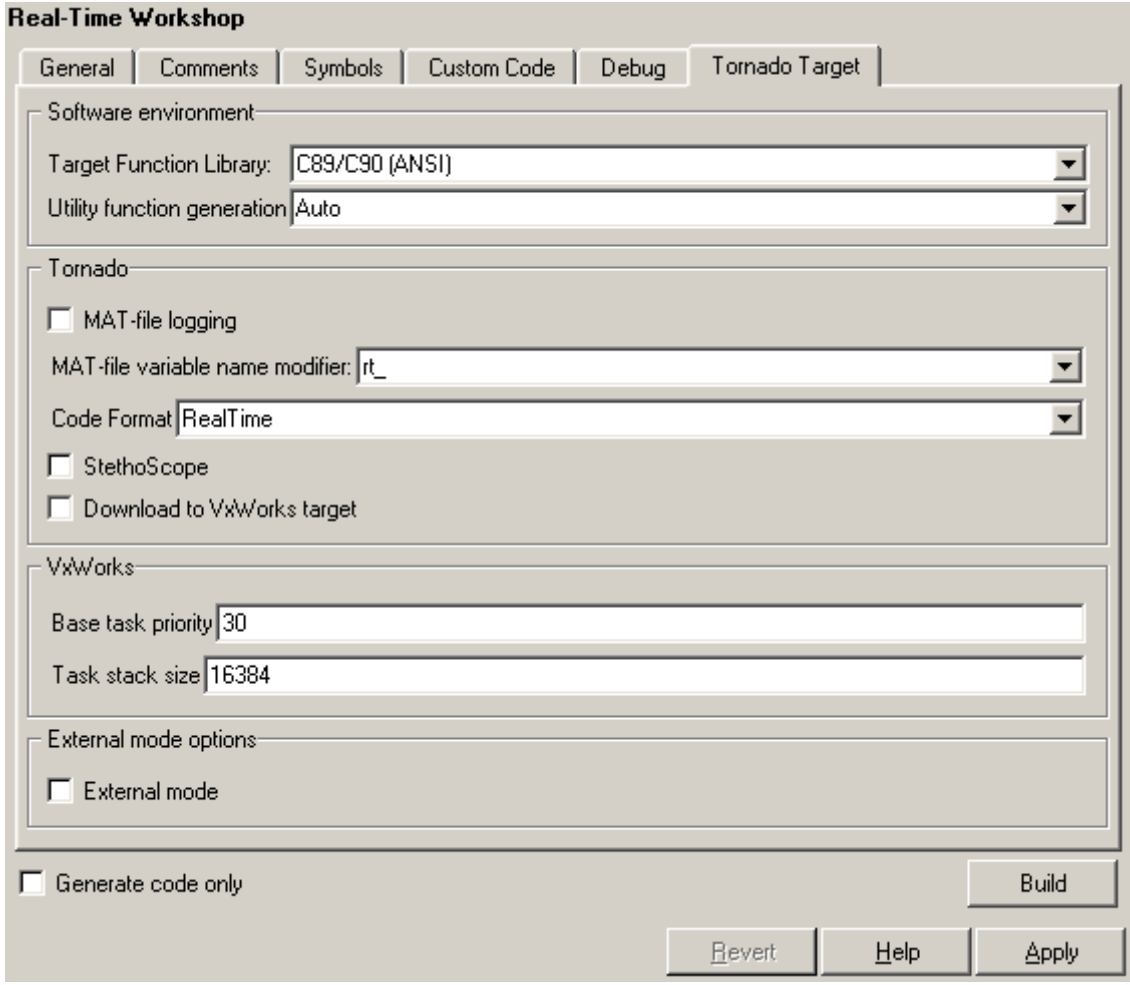
Command-Line Information

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

See Also

The S-Function Target

Real-Time Workshop Pane: Tornado Target



In this section...

In this section...

Tornado Target Tab Overview

Control the code generated by Real-Time Workshop for the Tornado Target.

Configuration

This tab appears only if you specify `tornado.tlc` as the System target file.

See Also

- *Tornado User's Guide* from Wind River Systems
- *StethoScope User's Guide* from Wind River Systems
- Targeting Tornado for Real-Time Applications

Target function library

Specify a target-specific 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.

`GNU99 (GNU)`
: Generates calls to the GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

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: `GenFloatMathFcnCalls`
Type: string
Value: `'ANSI_C' | 'C99 (ISO)' | 'GNU99 (GNU)'`
Default: `'ANSI_C'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Any valid library
Safety precaution	No impact

See Also

Configuring Model Interfaces

Utility function generation

Specify the location for generating utility functions.

Settings

Default: `Auto`

`Auto`
: Operates as follows:

- When the model contains Model blocks, place utilities within the `slprj/target/_sharedutils` directory.
- When the model does not contain Model blocks, place utilities in the build directory (generally, in *`model`*`.c` or *`model`*`.cpp`).

`Shared location`
: Directs code for utilities to be placed within the `slprj` directory in your working directory.

Command-Line Information

Parameter: `UtilityFuncGeneration`
Type: string
Value: `'Auto' | 'Shared location'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	`Shared location`
Traceability	`Shared location`
Efficiency	`Shared location`
Safety precaution	No impact

See Also

Configuring Model Interfaces

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: off

 On

Enables MAT-file logging. When you select this option, the code generator saves system states, output, and simulation time at each model execution time step. The data is written to a MAT-file, named (by default) *`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 needed 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

This parameter only appears for ERT-based targets and the Tornado target.

Command-Line Information

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

Recommended Settings

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

See Also

Using 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

When an ERT target is selected, 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

Data Logging

Code Format

Specify the code generation format.

Settings

Default: `RealTime`

`RealTime`
: Specifies the Real-Time code generation format.

`RealTimeMalloc`
: Specifies the Real-Time Malloc code generation format.

Command-Line Information

Parameter: `CodeFormat`
Type: string
Value: `'RealTime' | 'RealTimeMalloc'`
Default: `'RealTime'`

Recommended Settings

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

See Also

Targeting Tornado for Real-Time Applications

StethoScope

Specify whether to enable StethoScope, an optional data acquisition and data monitoring tool.

Settings

Default: off

 On

Enables StethoScope.

 Off

Disables StethoScope.

Tips

You can optionally monitor and change the parameters of the executing real-time program using either StethoScope or Simulink external mode, but not both with the same compiled image.

Dependencies

Enabling **StethoScope** automatically disables **External mode**, and vice versa.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact

Application	Setting
Efficiency	Off
Safety precaution	Off

See Also

- *Tornado User's Guide* from Wind River Systems
- *StethoScope User's Guide* from Wind River Systems
- Targeting Tornado for Real-Time Applications
- StethoScope Tasks
- StethoScope Monitoring

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, as described in Configuring for Automatic Downloading.
- Before every build, reset VxWorks by pressing **Ctrl+X** on the host console or power-cycling the VxWorks chassis. This ensures that no dangling processes or stale data exist in VxWorks when the automatic download occurs.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	Off

See Also

- *Tornado User's Guide* from Wind River Systems
- Targeting Tornado for Real-Time Applications
- Configuring for Automatic Downloading
- Building the Application
- Automatic Download and Execution

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, Real-Time Workshop 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: any valid value
Default: `30`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	May affect efficiency, depending on other task's priorities
Safety precaution	No impact

See Also

- *Tornado User's Guide* from Wind River Systems

- Targeting Tornado for Real-Time Applications

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: any 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
- Targeting Tornado for Real-Time Applications
- Task Stack Size

External mode

Specify whether to enable communication between Simulink and an application based on a client/server architecture.

Settings

Default: on

 On

Enables external mode. The client (Simulink) transmits messages requesting the server (application) to accept parameter changes or to upload signal data. The server responds by executing the request.

 Off

Disables external mode.

Dependencies

Selecting this parameter enables:

- **Transport layer**
- **MEX-file arguments**
- **Static memory allocation**

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

External Mode

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`.

Tips

- The MEX-file is specified in `extmode-transports.m`.
- The MEX-file cannot be edited.

Dependency

This parameter is enabled by **External Mode**.

Command-Line Information

Parameter: `ExtModeTransport`
Type: integer
Value: `0 | 1`
Default: `0`

Recommended Settings

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

See Also

- Creating an External Mode Communication Channel
- 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 **External Mode**.

Command-Line Information

Parameter: `ExtModeMexArgs`
Type: string
Value: any valid arguments
Default: `""`

Recommended Settings

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

See Also

- Target Interfacing
- Client/Server Implementations

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 **External Mode**.
- 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

External Mode Interface Options

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: any valid value
Default: 1000000

Recommended Settings

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

See Also

External Mode Interface Options

Parameter Reference

In this section...

Recommended Settings Summary

The following table summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the default (factory) configuration settings for the GRT target. For additional details, click the links in the Configuration Parameter column.

Mapping of Application Requirements to Configuration Parameters

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Default
"Solver Pane"					
Start Time	No impact	No impact	No impact	No impact	0.0 seconds
Stop time	No impact	No impact	No impact	No impact	10.0 seconds
Type	No impact	No impact	No impact	Fixed step	Variable step
Tasking mode for periodic sample times	No impact	No impact	No impact	No impact	Auto
"Data Import/Export Pane"					
Save to workspace	Maybe	Maybe	Off	No impact	On
Save options	Maybe	Maybe	No impact	No impact	On

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Default
"Optimization Pane"					
Block reduction	Off	Off	On	Off	On
Implement logic signals as Boolean data (vs. double)	No impact	No impact	On	On	On
Inline parameters	Off	On	On	No impact	Off
Conditional input branch execution	No impact	On	On	Off	On
Signal storage reuse	Off	Off	On	No impact	On
Application lifespan (days)	No impact	No impact	Finite value	`inf`	`inf`
Enable local block outputs	Off	No impact	On	No impact	On
Ignore integer downcasts in folded expressions	Off	No impact	On	Off	Off
Eliminate superfluous temporary variables (Expression folding)	Off	No impact	On	No impact	On
Loop unrolling threshold	No impact	No impact	>0	>1	5

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Default
Reuse block outputs	Off	Off	On	No impact	On
Inline invariant signals	Off	Off	On	No impact	Off
Remove code from floating-point to integer conversions that wraps out-of-range values	Off	Off	On	Off	Off
"Diagnostics Pane: Data Validity"					
Model Verification block enabling	No impact	No impact	No impact	No impact	Use local settings
"Hardware Implementation Pane"					
Device vendor	No impact	No impact	No impact	No impact	No impact
, **Device type**	No impact	No impact	No impact	No impact	No impact
Number of bits	No impact	No impact	Target specific	No impact	8, 16, 32, 32, 32
Byte ordering	No impact	No impact	No impact	No impact	Unspecified
Signed integer division rounds to	No impact	No impact	No impact	No impact	Undefined

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Default
Shift right on a signed integer as arithmetic shift	No impact	No impact	On	No impact	On
Emulation hardware (code generation only)	No impact	No impact	No impact	No impact	On
"Real-Time Workshop Pane: General" on page 6-3					
System target file	No impact	No impact	No impact	No impact	`grt.tlc`
Language	No impact	No impact	No impact	No impact	C
Generate HTML report	On	On	No impact	On	Off
Launch report automatically	On	On	No impact	No impact	Off
Compiler optimization level	Off	Off	On	No impact	Off
TLC options	No impact	No impact	No impact	No impact	`''`
Generate makefile	No impact	No impact	No impact	No impact	On
Make command	No impact	No impact	No impact	No impact	`make_rtw`
Template makefile	No impact	No impact	No impact	No impact	`grt_default_tmf`
Generate code only	Off	No impact	No impact	No impact	Off
"Real-Time Workshop Pane: Comments" on page 6-34					

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Default
Include comments	On	On	No impact	On	On
Simulink block 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
"Real-Time Workshop Pane: Symbols" on page 6-52					
Maximum identifier length	Any valid value	>30	No impact	>30	31
"Real-Time Workshop Pane: Custom Code" on page 6-84					
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 of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Default
"Real-Time Workshop Pane: Debug" on page 6-94					
Verbose build	On	No impact	No impact	On	On
Retain .rtw file	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
"Real-Time Workshop Pane: Interface" on page 6-103					
Target function library	No impact	No impact	Any valid value	No impact	`C89/C90 (ANSI)`
Utility function generation	Shared location	Shared location	Shared location	No impact	`Auto`
MAT-file variable name modifier	No impact	No impact	No impact	No impact	`rt_`
Interface	No impact	No impact	No impact	No impact	None
Signals in C API	No impact	No impact	No impact	No impact	On
Parameters in C API	No impact	No impact	No impact	No impact	On
Transport layer	No impact	No impact	No impact	No impact	`tcpip`

Mapping of Application Requirements to Configuration Parameters (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Default
MEX-file arguments	No impact	No impact	No impact	No impact	" "
Static memory allocation	No impact	No impact	No impact	No impact	Off

Parameter Command-Line Information Summary

The following table lists Real-Time Workshop® and Real-Time Workshop Embedded 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. For descriptions of the panes and options in that dialog box, see Configuration Parameters in the Real-Time Workshop documentation and "Configuration Parameters" in the Real-Time Workshop Embedded Coder documentation.

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 Real-Time Workshop Embedded Coder also provides buttons and scripts for customizing code generation.

For information about Simulink® parameters, see "Configuration Parameters Dialog Box" in the Simulink documentation. For information on using `get_param` and `set_param` to tune the parameters for various model configurations, see "Parameter Tuning by Using MATLAB Commands". See "Using Configuration Wizard Blocks" in the Real-Time Workshop Embedded Coder documentation for information on using Configuration Wizard features.

Note Parameters that are specific to the ERT target or targets based on the ERT target, Stateflow®, or Fixed-Point Toolbox support are marked with (ERT), (Stateflow), and (Fixed-Point), respectively. To set the values of parameters marked with (ERT), you must specify an ERT or ERT-based target for your configuration set. Also, note that the default setting for a parameter might vary for different targets. Parameters marked with (ERT) are listed with ERT target defaults.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
Optimization pane		
`BufferReuse` `off`, **`on`**	**Optimization > Reuse block outputs**	Reuse local (function) variables for block outputs wherever possible. Selecting this option trades code traceability for code efficiency.
`DataBitsets` (Stateflow) **`off`**, `on`	**Optimization > Use bit sets for storing boolean data**	Use bit sets for storing Boolean data.
`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.
`EnforceIntegerDowncast` `off`, **`on`**	**Optimization > Ignore integer downcasts in folded expressions**	Remove casts of intermediate variables to improve code efficiency. When you select this option, expressions involving 8-bit and 16-bit arithmetic on microprocessors of a larger bit size are less likely to overflow in code than in simulation.
`ExpressionFolding` `off`, **`on`**	**Optimization > Eliminate superfluous temporary variables (Expression folding) > Interface**	Collapse block computations into single expressions wherever possible. This improves code readability and efficiency.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`InitFltsAndDblsToZero` (ERT) `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.
`InlineInvariantSignals` `off`, **`on`**	**Optimization > Inline invariant signals**	Precompute and inline the values of invariant signals in the generated code.
`InlinedParameterPlacement` (ERT) `Hierarchical`, **`NonHierarchical`**	**Optimization > Parameter structure**	Specify how generated code stores global (tunable) parameters. Specify `NonHierarchical` to trade off modularity for efficiency.
`LifeSpan` (ERT) *`string`*	**Optimization > Application lifespan (days)**	Optimize the size of counters used to compute absolute and elapsed time, using the specified application life span value.
`LocalBlockOutputs` `off`, **`on`**	**Optimization > Enable local block outputs**	Declare block outputs in local (function) scope wherever possible to reduce global RAM usage.
`NoFixptDivByZeroProtection` (Fixed-Point Toolbox) **`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.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`OptimizeModelRefInitCode` (ERT) **`off`**, `on`	**Optimization > Optimize initialization code for model reference**	Suppress generation of initialization code to accommodate the case where this model is referred to by a subsystem that resets its states when enabled. Select this option if the model will never be referred to by such a subsystem. Simulink reports an error if this constraint is violated, in which case you can disable this optimization.
`RollThreshold` `int - 5`	**Optimization > Loop unrolling threshold**	Specify the minimum signal width for which a `for` loop is to be generated.
`StateBitsets` (Stateflow) **`off`**, `on`	**Optimization > Use bit sets for storing state configuration**	Use bit sets for storing state configuration.
`UseTempVars` (Stateflow) **`off`**, `on`	**Optimization > Minimize array reads using temporary variables**	Minimize array reads in global memory by using temporary variables.
`ZeroExternalMemoryAtStartup` (ERT) `off`, **`on`**	**Optimization > Remove root level I/O zero initialization**	Suppress code that initializes root-level I/O data structures to zero.
`ZeroInternalMemoryAtStartup` (ERT) `off`, **`on`**	**Optimization > Remove internal state zero initialization**	Suppress code that initializes global data structures (for example, block I/O data structures) to zero.
Diagnostics panes		

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ParameterTunabilityLossMsg` none, **`warning`**, `error`	**Diagnostics > Data Validity > Detect Loss of Tunability**	Specifies diagnostic action to take when a parameter cannot be tuned because it uses unsupported functions or operators.
`SignalLabelMismatchMsg` **`None`**, `Warning`, `Error`	**Diagnostics > Connectivity > Signal label mismatch**	Specify the diagnostic action to take when a signal label mismatch occurs.
Hardware Implementation pane		
`ProdHWDeviceType` *`string`* - `32-bit Generic`	**Hardware Implementation > Emulation hardware > Device vendor** combined with **Device type**	Specify a string of the form *`vendor->type`* that selects a device vendor and a device type among the values listed in the **Device vendor** and **Device type** drop-down menus. For example, `Analog Devices->Blackfin`.
`TargetBitPerChar` `int` - **`8`**	**Hardware Implementation > Emulation hardware > char**	Specify the number of bits used to represent the C/C++ type `char`.
`TargetBitPerInt` `int` - **`32`**	**Hardware Implementation > Emulation hardware > int**	Specify the number of bits used to represent the C/C++ type `int`.
`TargetBitPerLong` `int` - **`32`**	**Hardware Implementation > Emulation hardware > long**	Specify the number of bits used to represent the C/C++ type `long`.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TargetBitPerShort` `int` - **16**	**Hardware Implementation > Emulation hardware > short**	Specify the number of bits used to represent the C/C++ type `short`.
`TargetEndianess` **`Unspecified`**, `LittleEndian`, `BigEndian`	**Hardware Implementation > Emulation hardware > Byte ordering**	Specify whether the byte ordering of the target is Big Endian (most significant byte first) or Little Endian (least significant byte first). If left unspecified, Real-Time Workshop generates executable code to compute the result.
`TargetIntDivRoundTo` `Zero`, `Floor`, **`Undefined`**	**Hardware Implementation > Emulation hardware > Signed integer division rounds to**	Specify how your C/C++ compiler rounds the result of dividing two signed integers. This information enables the code generator to generate efficient C or C++ code from the model.
`TargetShiftRightIntArith` `off`, **`on`**	**Hardware Implementation > Emulation hardware > Shift right on a signed integer as arithmetic shift**	Specify that your C/C++ compiler implements a right shift of a signed integer as an arithmetic right shift. Virtually all compilers do this.
`TargetWordSize` `int` - **32**	**Hardware Implementation > Emulation hardware > native word size**	Specify the number of bits that the target processor can process at one time. Providing the processor's native word size allows for more efficient code to be generated when converting the endian byte order of data types.
Real-Time Workshop pane: General tab		

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenCodeOnly` *`string`* - **`off`**, `on`	**Real-Time Workshop > General > Generate code only**	Generate source code, but do not execute the makefile to build an executable.
`GenerateMakefile` *`string`* - `off`, **`on`**	**Real-Time Workshop > General > Generate makefile**	Specify whether Real-Time Workshop is to generate a makefile during the build process for a model.
`GenerateReport` `string` - **`off`**, `on`	**Real-Time Workshop > General > Generate HTML report**	Document the generated C or C++ code in an HTML report.
`GenerateTraceInfo` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > General > Block-to-code highlighting**	Includes block-to-code highlighting support in the generated HTML report.
`IgnoreCustomStorageClasses` (ERT) *`string`* - `off`, **`on`**	**Real-Time Workshop > General > Ignore custom storage classes**	Treat custom storage classes as `'Auto'`.
`IncludeHyperlinkInReport` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > General > Code-to-block highlighting**	Link code segments to the corresponding block in the model. This option increases code generation time for large models.
`LaunchReport` `string` - **`off`**, `on`	**Real-Time Workshop > General > Launch report automatically**	Display the HTML report after code generation completes.
`MakeCommand` *`string`* - **`make_rtw`**	**Real-Time Workshop > General > Make command**	Specify the `make` command and optional arguments to be used to generate an executable for the model.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`RTWCompilerOptimization` string - **`Off`**, `On`, `Custom`	**Real-Time Workshop > General > Compiler optimization level**	Use this parameter to trade off compilation time against run time for your model code without having to supply compiler-specific flags to other levels of the Real-Time Workshop 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	**Real-Time Workshop > General > 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`	**Real-Time Workshop > General > Save build log**	Save build log.
`SystemTargetFile` `string` - **`grt.tlc`**	**Real-Time Workshop > General > System target file**	Specify a system target file.
`TargetLang` `string` - **`C`**, `C++`	**Real-Time Workshop > General > Language**	Specify whether Real-Time Workshop is to generate C or C++ code.
`TemplateMakefile` *`string`* - **`grt_default_tmf`**	**Real-Time Workshop > General > Template makefile**	Specify the current template makefile for building a Real-Time Workshop target.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TLCOptions` *string* - `''`	**Real-Time Workshop > General > TLC options**	Specify additional TLC command line options.
Real-Time Workshop pane: Comments tab		
`CustomCommentsFcn` (ERT) *string* - `''`	**Real-Time Workshop > Comments > Custom comments function**	Specify the filename of the M-function or TLC function that adds the custom comment.
`EnableCustomComments` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Comments > Custom comments (MPT objects only)**	Add a comment above a signal's or parameter's identifier in the generated file.
`ForceParamTrailComments` *string* - **`off`**, `on`	**Real-Time Workshop > 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`**	**Real-Time Workshop > Comments > Include comments**	Include comments in generated code.
`InsertBlockDesc` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Comments > Simulink block descriptions**	Insert the contents of the **Description** field from the Block Parameters dialog box into the generated code as a comment.
`ReqsInCode` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Comments > Requirements in block comments**	Include specified requirements in the generated code as a comment.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SFDataObjDesc` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Stateflow object descriptions**	Insert Stateflow object descriptions into the generated code as a comment.
`ShowEliminatedStatements` *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Show eliminated blocks**	Show statements for eliminated blocks as comments in the generated code.
`SimulinkBlockComments` *`string`* - `off`, **`on`**	**Real-Time Workshop > Comments > Simulink block comments**	Insert Simulink block names as comments above the generated code for each block.
`SimulinkDataObjDesc` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Simulink data object descriptions**	Insert Simulink data object descriptions into the generated code as comments.
Real-Time Workshop pane: Symbols tab		
`CustomSymbolStrBlkIO` (ERT) *`string`* - **`rtb_$N$M`**	**Real-Time Workshop > Symbols > Local block output variables**	Specify a symbol format rule for local block output variables. The rule can contain valid C identifier characters and the following macros: $M - Mangle $N - Name of object $A - Data type acronym

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSymbolStrFcn` (ERT) *`string`* - **`$R$N$M$F`**	**Real-Time Workshop > Symbols > Subsystem methods**	Specify a symbol format rule for subsystem methods. The rule can contain valid C identifier characters and the following macros: $M - Mangle $R - Root model name $N - Name of object $H - System hierarchy number $F - Subsystem method name
`CustomSymbolStrField` (ERT) *`string`* - **`$N$M`**	**Real-Time Workshop > Symbols > Field name of global types**	Specify a symbol format rule for field name of global types. The rule can contain valid C identifier characters and the following macros: $M - Mangle $N - Name of object $H - System hierarchy number $A - Data type acronym
`CustomSymbolStrGlobalVar` (ERT) *`string`* - **`$R$N$M`**	**Real-Time Workshop > Symbols > Global variables**	Specify a symbol format rule for global variables. The rule can contain valid C identifier characters and the following macros: $M - Mangle $R - Root model name $N - Name of object
`CustomSymbolStrMacro` (ERT) *`string`* - **`$R$N$M`**	**Real-Time Workshop > Symbols > Constant macros**	Specify a symbol format rule for constant macros. The rule can contain valid C identifier characters and the following macros: $M - Mangle $R - Root model name $N - Name of object

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSymbolStrTmpVar` (ERT) *`string`* - **`$N$M`**	**Real-Time Workshop > Symbols > Local temporary variables**	Specify a symbol format rule for local temporary variables. The rule can contain valid C identifier characters and the following macros: $M - Mangle $R - Root model name $N - Name of object
`CustomSymbolStrType` (ERT) *`string`* - **`$N$R$M`**	**Real-Time Workshop > Symbols > Global types**	Specify a symbol format rule for global types. The rule can contain valid C identifier characters and the following macros: $M - Mangle $R - Root model name $N - Name of object
`DefineNamingFcn` *`string`* -`''`	**Real-Time Workshop > Symbols > #define naming > Custom M-function**	Specify a custom M-function to control the naming of symbols with `#define` statements. You can set this parameter only if `DefineNamingRule` is set to `Custom`.
`DefineNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > #define naming**	Specify the rule that changes the spelling of all `#define` names.
`IncDataTypeInIds` **`off`**, `on`	**Real-Time Workshop > Symbol > Include data type acronym in identifiers**	Include acronyms that express data types in signal and work vector identifiers. For example, `'rtB.i32_signame'` identifies a 32-bit integer block output signal named `'signame'`.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IncHierarchyInIds` **`off`**, `on`	**Real-Time Workshop > Symbols > Include system hierarchy number in identifiers**	Include the system hierarchy number in variable identifiers. For example, `'s3_'` is the system hierarchy number in `rtB.s3_signame` for a block output signal named `'signame'`. Including the system hierarchy number in identifiers improves the traceability of generated code. To locate the subsystem in which the identifier resides, type `hilite_system('<S3>')` at the MATLAB prompt. The argument specified with `hilite_system` requires an uppercase `S`.
`InlinedPrmAccess` (ERT) string - **`Literals`**, `Macros`	**Real-Time Workshop > Symbols > Generate scalar inlined parameters as**	Specify whether inlined parameters are coded as numeric constants or macros. Specify `Macros` for more efficient code.
`MangleLength` `int` - **`1`**	**Real-Time Workshop > Symbols > Minimum mangle length**	Specify the minimum number of characters to be used for name mangling strings generated and applied to symbols to avoid name collisions. A larger value reduces the chance of identifier disturbance when you modify the model.
`MaxIdLength` `int` - **`31`**	**Real-Time Workshop > Symbols > Maximum identifier length**	Specify the maximum number of characters that can be used in generated function, type definition, and variable names.
`ParamNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > Parameter naming**	Select a rule that changes spelling of all parameter names.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`PrefixModelToSubsysFcnNames` `off`, **`on`**	**Real-Time Workshop > Symbols > Prefix model name to global identifiers**	Add the model name as a prefix to subsystem function names for all code formats. When appropriate for the code format, also add the model name as a prefix to top-level functions and data structures. This prevents compiler errors due to name clashes when combining multiple models.
`SignalNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > Signal naming**	Specify a rule the code generator is to use that changes spelling of all signal names.
Real-Time Workshop pane: Custom Code tab		
`CustomHeaderCode` *`string`* - `''`	**Real-Time Workshop > Custom Code > Header file**	Specify the code to appear at the top of the generated *`model`*`.h` header file.
`CustomInclude` *`string`* - `''`	**Real-Time Workshop > Custom Code > Include directories**	Specify a space-separated list of include directories to be added to the include path when compiling the generated code.
`CustomInitializer` *`string`* - `''`	**Real-Time Workshop > Custom Code**	Specify the code to appear in the generated model initialize function.
`CustomLibrary` *`string`* - `''`	**Real-Time Workshop > Custom Code > Initialize function Libraries**	Specify a space-separated list of static library files to be linked with the generated code.
`CustomSource` *`string`* - `''`	**Real-Time Workshop > Custom Code > Source files**	Specify a space-separated list of source files to be compiled and linked with the generated code.
`CustomSourceCode` *`string`* - `''`	**Real-Time Workshop > Custom Code > Source file**	Specify code to appear at the top of the generated *`model`*`.c` source file.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomTerminator` *`string`* - `''`	**Real-Time Workshop > Custom Code > Terminate function**	Specify code to appear in the model's generated terminate function.
Real-Time Workshop pane: Debug tab		
`ProfileTLC` string - **`off`**, `on`	**Real-Time Workshop > 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`**	**Real-Time Workshop > Debug > Verbose build**	Display messages indicating code generation stages and compiler output.
`RetainRTWFile` string - **`off`**, `on`	**Real-Time Workshop > Debug > Retain .rtw file**	Retain the *`model`*`.rtw` file in the current build directory.
`TLCAssert` string - **`off`**, `on`	**Real-Time Workshop > Debug > Enable TLC assertion**	Produce a TLC stack trace when the argument to the `assert` directives evaluates to `false`.
`TLCCoverage` string - **`off`**, `on`	**Real-Time Workshop > 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`	**Real-Time Workshop > 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.
Real-Time Workshop pane: Interface tab		

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CombineOutputUpdateFcns` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > Single output/update function**	Generate a model's output and update routines into a single-step function.
`ExtMode` **`off`**, `on`	**Real-Time Workshop > Interface > Interface**	Specify the data interface to be generated with the code.
`ExtModeMexArgs` *`string`* (`""`)	**Real-Time Workshop > Interface > Interface > External > MEX-file arguments**	Specify arguments that are passed to an external mode interface MEX-file for communicating with executing targets.
`ExtModeStaticAlloc` **`off`**, `on`	**Real-Time Workshop > Interface > Static memory allocation**	Use a static memory buffer for external mode instead of allocating dynamic memory (calls to `malloc`).
`ExtModeStaticAllocSize` *`integer`* (**`1000000`**)	**Real-Time Workshop > 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 32-bit Windows serial	**Real-Time Workshop > Interface > Interface > External > Transport layer**	Specify transport protocols for external mode communications.
`GenerateASAP2` **`off`**, `on`	**Real-Time Workshop > Interface > Interface**	Specify the data interface to be generated with the code.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateErtSFunction` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Create Simulink (S-Function) block**	Wrap the generated code inside an S-Function block. This allows you to validate the generated code in Simulink.
`GenFloatMathFcnCalls` string - **`ANSI_C`**, `C99 (ISO)`, `GNU99 (GNU)` (For ERT-based models, additional target-specific values may be available; see the **Target function library** drop-down list in the Configuration Parameters dialog box.)	**Real-Time Workshop > Interface > Target function library**	Specify a target-specific math library for your model. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur. `ANSI_C` - ISO/IEC 9899:1990 C standard math library for floating-point functions `C99 (ISO)` - ISO/IEC 9899:1999 C standard math library `GNU99 (GNU)` - GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`
`GRTInterface` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > GRT compatible call interface**	Include a code interface (wrapper) that is compatible with the GRT target.
`IncludeMdlTerminateFcn` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Terminate function required**	Generate a terminate function for the model.
`LogVarNameModifier` string - **`none`**, `rt_`, `_rt`	**Real-Time Workshop > Interface > MAT-file variable name modifier**	Augment the MAT-file variable name.
`MatFileLogging` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > MAT-file logging**	Generate code that logs data to a MATLAB `.mat` file.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MultiInstanceErrorCode` (ERT) string - `None`, `Warning`, **`Error`**	**Real-Time Workshop > Interface > Reusable code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of reusable code.
`MultiInstanceERTCode` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Reusable code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of reusable code.
`PortableWordSizes` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Enable portable word sizes**	Specify that model code should be generated with conditional processing macros that allow the same generated source code files to be used both for software-in-the-loop (SIL) testing on the host platform and for production deployment on the target platform.
`PurelyIntegerCode` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > floating-point numbers**	Support floating-point data types in the generated code. This option is forced on when `SupportNonInlinedSFcns` is on.
`RTWCAPIParams` string - **`off`**, `on`	**Real-Time Workshop > Interface > Parameters in C API**	Generate parameter tuning structures in C API.
`RTWCAPISignals` string - **`off`**, `on`	**Real-Time Workshop > Interface > Signals in C API**	Generate signal structure in C API.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`RootIOFormat` (ERT) string - **`Individual arguments`**, `Structure reference`	**Real-Time Workshop > Interface > Pass root-level I/O as**	Specify how the code generator is to pass root-level I/O data into a reusable function.
`SupportAbsoluteTime` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > absolute time**	Support absolute time in the generated code. Blocks such as the Discrete Integrator might require absolute time.
`SupportComplex` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > complex numbers**	Support complex data types in the generated code.
`SupportContinuousTime` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > continuous time**	Support continuous time in the generated code. This allows blocks to be configured with a continuous sample time. Not available if `SuppressErrorStatus` is `on`.
`SupportNonFinite` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > nonfinite numbers**	Support nonfinite values (`inf`, `nan`, `-inf`) in the generated code. This option is forced on when `SupportNonInlinedSFcns` is on.
`SupportNonInlinedSFcns` string - `off`, **`on`**	**Real-Time Workshop > Interface > noninlined S-functions**	Support S-functions that have not been inlined with a TLC file. Inlined S-functions generate the most efficient code.
`SuppressErrorStatus` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Suppress error status in real-time model data structure**	Remove the error status field of the real-time model data structure to preserve memory. When `on`, `SupportContinuousTime` is `off`.
`UtilityFuncGeneration` string - **`Auto`**, `Shared location`	**Real-Time Workshop > Interface > Utility function generation**	Specify where utility functions are to be generated.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
Real-Time Workshop pane: Code Style tab		
`ParenthesesLevel` (ERT) string - `Minimum`, **`Nominal`**, `Maximum`	**Real-Time Workshop > Code Style > Parentheses Level**	Control existence of optional parentheses in generated code.
`PreserveExpressionOrder` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve operand order in expression**	Control reordering of commutable expressions.
`PreserveIfCondition` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve condition expression in if statement**	Control preservation of `if` statement conditions.
Real-Time Workshop pane: Templates tab		
`ERTCustomFileTemplate` (ERT) *`string`* - **`example_file_process.tlc`**	**Real-Time Workshop > Templates > File customization template**	Specify a TLC callback script for customizing the generated code.
`ERTDataHdrFileTemplate` (ERT) *`string`* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Header file (*.h) template**	Specify a template that organizes the generated data `.h` header files.
`ERTDataSrcFileTemplate` (ERT) *`string`* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Source file (*.c or *.cpp) template**	Specify a template that organizes the generated data `.c` source files.
`ERTHdrFileBannerTemplate` (ERT) *`string`* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Header file (*.h) template**	Specify a template that organizes the generated code `.h` header files.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ERTSrcFileBannerTemplate` (ERT) *string* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Source file (*.c or *.cpp) template**	Specify a template that organizes the generated code `.c` or `.cpp` source files.
`GenerateSampleERTMain` (ERT) string - **`off`**, on	**Real-Time Workshop > Templates > Generate an example main program**	Generate an example main program that demonstrates how to deploy the generated code. The program is written to the file `ert_main.c` or `ert_main.cpp`.
`TargetOS` (ERT) string - **`BareBoardExample`**, `VxWorksExample`	**Real-Time Workshop > Templates > Target operating system**	Specify the target operating system for the example main `ert_main.c` or `ert_main.cpp`. `BareBoardExample` is a generic example that assumes no operating system. `VxWorksExample` is tailored to the VxWorks real-time operating system.
Real-Time Workshop pane: Data Placement tab		
`DataDefinitionFile` (ERT) *string* - **`global.c`**	**Real-Time Workshop > Data Placement > Data definition filename**	Specify the name of a single separate `.c` or `.cpp` file that contains global data definitions.
`DataReferenceFile` (ERT) *string* - **`global.h`**	**Real-Time Workshop > Data Placement > Data declaration filename**	Specify the name of a single separate `.c` or `.cpp` file that contains global data references.
`GlobalDataDefinition`(ERT) string - **`Auto`**, `InSourceFile`, `InSeparateSourceFile`	**Real-Time Workshop > Data Placement > Data definition**	Select the `.c` or `.cpp` file where variables of global scope are defined.
`GlobalDataReference` (ERT) string - **`Auto`**, `InSourceFile`, `InSeparateHeaderFile`	**Real-Time Workshop > Data Placement > Data declaration**	Select the `.h` file where variables of global scope are declared (for example, `extern real_T globalvar;`).

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IncludeFileDelimiter` (ERT) string - **`Auto`**, `UseQuote`, `UseBracket`	**Real-Time Workshop > Data Placement > #include file delimiter**	Specify the delimiter to be used for all data objects that do not have a delimiter specified in the `IncludeFile` property.
`ModuleName` (ERT) *`string`* - `''`	**Real-Time Workshop > Data Placement > Module name**	Specify the name of the module that owns this model.
`ModuleNamingRule` (ERT) string - **`Unspecified`**, `SameAsModel`, `UserSpecified`	**Real-Time Workshop > Data Placement > Module naming**	Specify the rule to be used for naming the module.
`ParamTuneLevel` (ERT) `int` - **`10`**	**Real-Time Workshop > Data Placement > Parameter tune level**	Specify whether the code generator is to declare a parameter data object as tunable global data in the generated code.
`SignalDisplayLevel` (ERT) `int` - **`10`**	**Real-Time Workshop > Data Placement > Signal display level**	Specify whether the code generator is to declare a signal data object as global data in the generated code.
Real-Time Workshop pane: Data Type Replacement tab		
`EnableUserReplacementTypes` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Data Type Replacement**	Specify whether to replace built-in data type names with user-defined data type names in generated code.
`ReplacementTypes` (ERT) *`string`* - `''`	**Real-Time Workshop > Data Type Replacement > Data type names**	Specify names to use for built-in data types in generated code.
Real-Time Workshop pane: Memory Sections tab		
`MemSecPackage` (ERT) string - **`--- None ---`**, `Simulink`, `mpt`	**Real-Time Workshop > Memory Sections > Package**	Specify the package that contains the memory sections that you want to apply.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MemSecFuncInitTerm` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Initialize/Terminate**	Apply memory sections to: • Initialize/Start functions • Terminate functions
`MemSecFuncExecute` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Execution**	Apply memory sections to: • Step functions • Run-time initialization functions • Derivative functions • Enable functions • Disable functions
`MemSecDataConstants` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Constants**	Apply memory sections to: • Constant parameters • Constant block I/O • Zero representation
`MemSecDataIO` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Inputs/Outputs**	Apply memory sections to: • Root inputs • Root outputs
`MemSecDataInternal` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Internal data**	Apply memory sections to: • Block I/O • D-work vectors • Run-time model • Zero-crossings

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MemSecDataParameters` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Parameters**	Apply memory sections to: • Parameters
Not in GUI		
`CodeGenDirectory`	Not available	For MathWorks use only.
`Comment`	Not available	For MathWorks use only.
`CompOptLevelCompliant` `off`, `on`	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to use the **Compiler optimization level** parameter on the **Interface** pane to control the compiler optimization level for building generated code. Default is `off` for custom targets and `on` for targets provided by Real-Time Workshop and Real-Time Workshop Embedded Coder.
`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.
`ERTFirstTimeCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to control inclusion of the `firstTime` argument in the *`model`*`_initialize` function generated for a Simulink model. Default is `off` for custom and non-ERT targets and `on` for ERT targets.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`EvaledLifeSpan`	Not available	For MathWorks use only.
`ExtModeMexFile`	Not available	For MathWorks use only.
`ExtModeTesting`	Not available	For MathWorks use only.
`FoldNonRolledExpr`	Not available	For MathWorks use only.
`GenerateFullHeader`	Not available	For MathWorks use only.
`IncAutoGenComments`	Not available	For MathWorks use only.
`IncludeERTFirstTime` (ERT) string - **`off`**, `on`	Not available	Specify whether Real-Time Workshop Embedded Coder is to include the `firstTime` argument in the *`model`*`_initialize` function generated for a Simulink model.
`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.
`IsPILTarget`	Not available	For MathWorks use only.
`ModelReferenceCompliant` string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports model reference.
`ModelStepFunctionPrototype` `ControlCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to control the function prototypes of step functions that are generated for a Simulink model. Default is `off` for non-ERT targets and `on` for ERT targets.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ParamNamingFcn`	Not available	For MathWorks use only.
`PostCodeGenCommand` *`string`* - `''`	Not available	Add the specified post code generation command to the model's 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.
`RTWCAPIStates`	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.)
`TargetPreCompLibLocation` *`string`* - `''`	Not available	Control the location of precompiled libraries. If you do not set this parameter, Real-Time Workshop uses the location specified in `rtwmakecfg.m`.
`TargetPreprocMaxBitsSint` int - **32**	Not available	Specify the maximum number of bits that the target C preprocessor can use for signed integer math.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TargetPreprocMaxBitsUint` int - **32**	Not available	Specify the maximum number of bits that the target C preprocessor can use for unsigned integer math.

`TargetTypeEmulationWarnSuppressLevel`
`SuppressLevel`
int - **0**
	Not available	When greater than or equal to 2, suppress warning messages that Real-Time Workshop displays when emulating integer sizes in rapid prototyping environments.

7

Embedded MATLAB Coder Configuration Parameters

Real-Time Workshop Dialog Box for Embedded MATLAB Coder

In this section...

Real-Time Workshop Dialog Box Overview

Specifies parameters for embeddable C code generation using Embedded MATLAB Coder.

Displaying the Dialog Box

To display the Real-Time Workshop dialog box for Embedded MATLAB Coder, follow these steps at the MATLAB command prompt:

1. Define a configuration object variable for embeddable C code generation in the MATLAB workspace by issuing a constructor command like this:

   ```
   codegen_cfg=emlcoder.RTWConfig;
   ```

2. Open the property dialog box using one of these methods:
   - Double-click the configuration object variable in the MATLAB workspace
   - Issue the `open` command from the MATLAB prompt, passing it the configuration object variable, as in this example:

     ```
     open codegen_cfg;
     ```

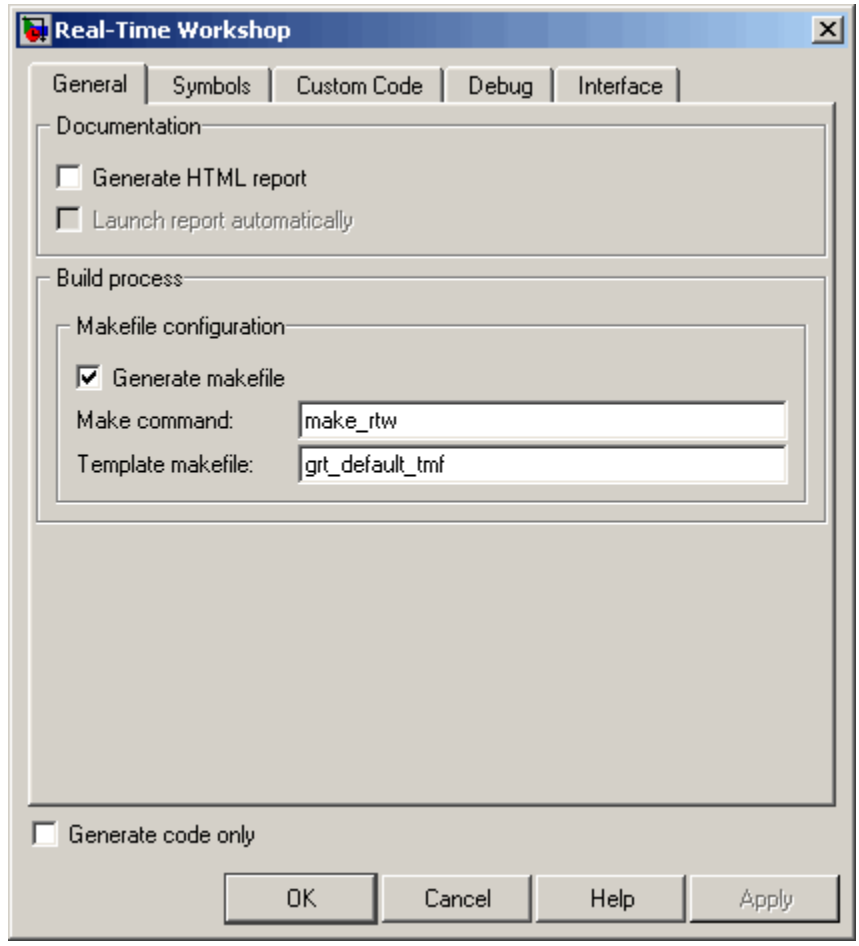
   The dialog box displays on your desktop.

See Also

"Configuring Your Environment for Code Generation"

General Tab

Specifies general parameters for embeddable C code generation using Embedded MATLAB Coder.



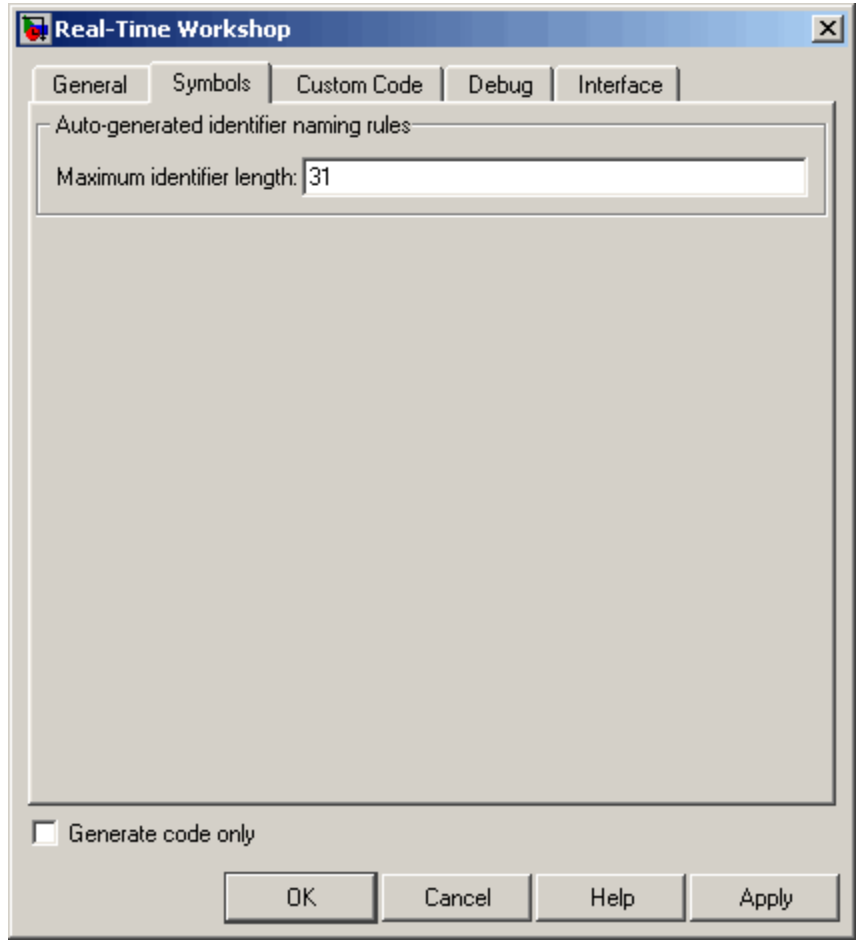
Parameters

The following table describes the general parameters for the Embedded MATLAB Coder Real-Time Workshop dialog box:

General Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Generate HTML report" on page 6-9	`GenerateReport` `true`, **`false`**	Document generated code in an HTML report.
"Launch report automatically" on page 6-11	`LaunchReport` `true`, **`false`**	Specify whether to automatically display HTML reports after code generation completes. **Note** Requires that you select **Generate HTML report**
"Generate makefile" on page 6-23	`GenerateMakefile` **`true`**, `false`	Specify whether to generate a makefile during the build process.
"Make command" on page 6-25	`MakeCommand` *`string`*, **`'make_rtw'`**	Specify a make command (if **Generate makefile** is enabled)
"Template makefile" on page 6-27	`MakeCommand` *`string`*, **`'grt_default_tmf'`**	Specify a template makefile (if **Generate makefile** is enabled)

Symbols Tab

Specifies parameters for selecting automatically generated naming rules for identifiers in code generation using Embedded MATLAB Coder.



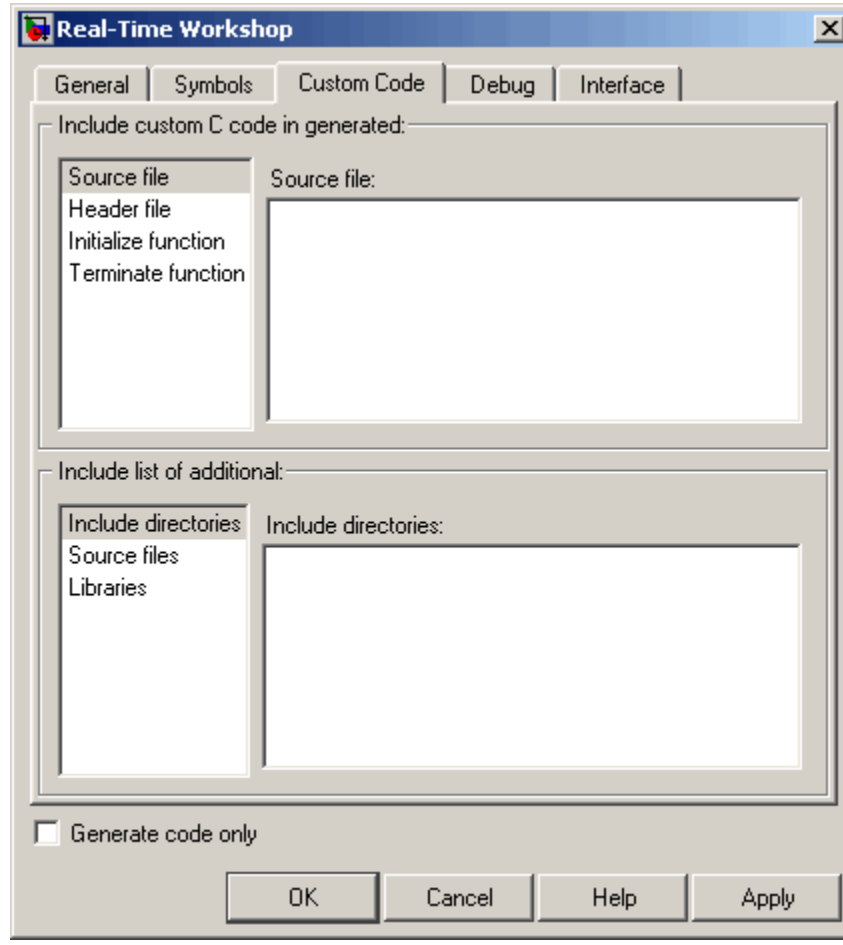
Parameters

The following table describes the symbols parameters for the Embedded MATLAB Coder Real-Time Workshop dialog box:

Symbols Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Maximum identifier length" on page 6-73	`MaxIdLength` *`integer`*, **`31`**	Specify maximum number of characters in generated function, type definition, and variable names. Minimum is 31.

Custom Code Tab

Creates a list of custom C code, directories, source and header files, and libraries to be included in files generated by Embedded MATLAB Coder.



Configuration

1. Select the type of information to include from the list on the left side of the pane.
2. Enter a string to identify the specific code, directory, source file, or library.
3. Click **Apply**.

Parameters

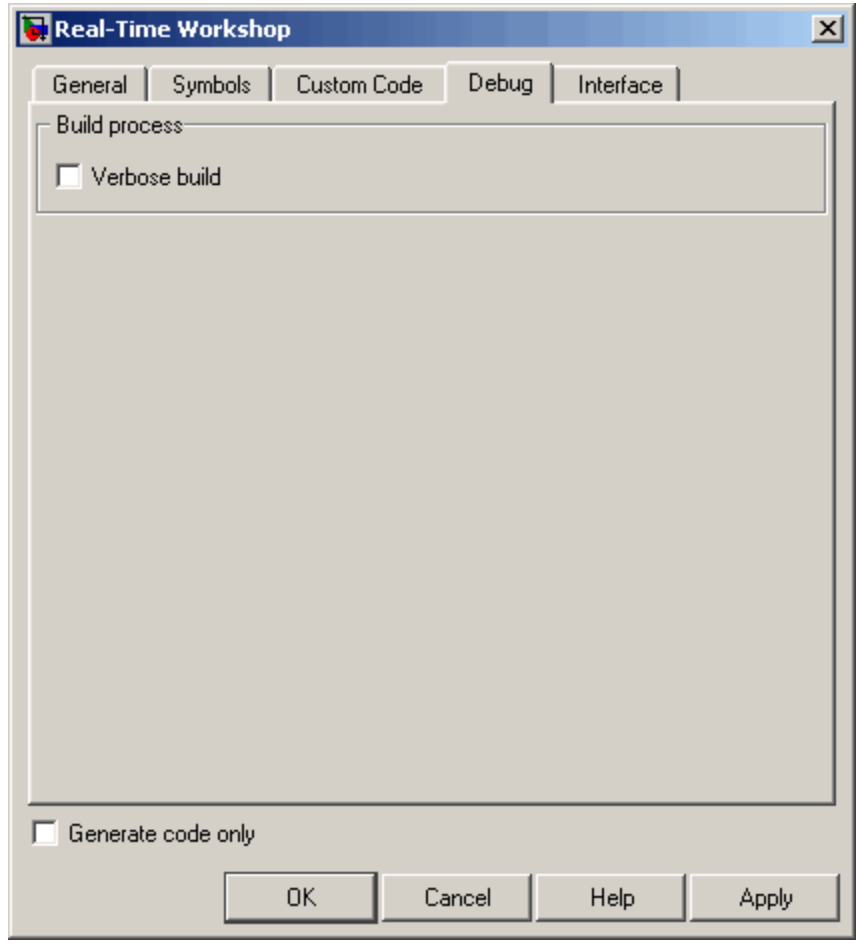
The following table describes the custom code parameters for the Embedded MATLAB Coder Real-Time Workshop dialog box:

Custom Code Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Source file" on page 6-87	`CustomSourceCode` *`string`*, **`''`**	Specify code appearing near the top of the generated `.c` or `.cpp` file, outside of any function.
"Header file" on page 6-88	`CustomHeaderCode` *`string`*, **`''`**	Specify code appearing near the top of the generated `.h` file.
"Initialize function" on page 6-89	`CustomInitializer` *`string`*, **`''`**	Specify code appearing in the initialize function of the generated `.c` or `.cpp` file.
"Terminate function" on page 6-90	`CustomTerminator` *`string`*, **`''`**	Specify code appearing in the terminate function of the generated `.c` or `.cpp` file.
"Include directories" on page 6-91	`CustomInclude` *`string`*, **`''`**	Specify a space-separated list of include directories to be added to the include path when compiling the generated code.

Custom Code Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Source files" on page 6-92	`CustomSource` *`string`*, `''`	Specify a space-separated list of source files to be compiled and linked with the generated code.
"Libraries" on page 6-93	`CustomLibrary` *`string`*, `''`	Specify a list of additional libraries to link with.

Debug Tab

Specifies parameters for debugging the Embedded MATLAB Coder build process.



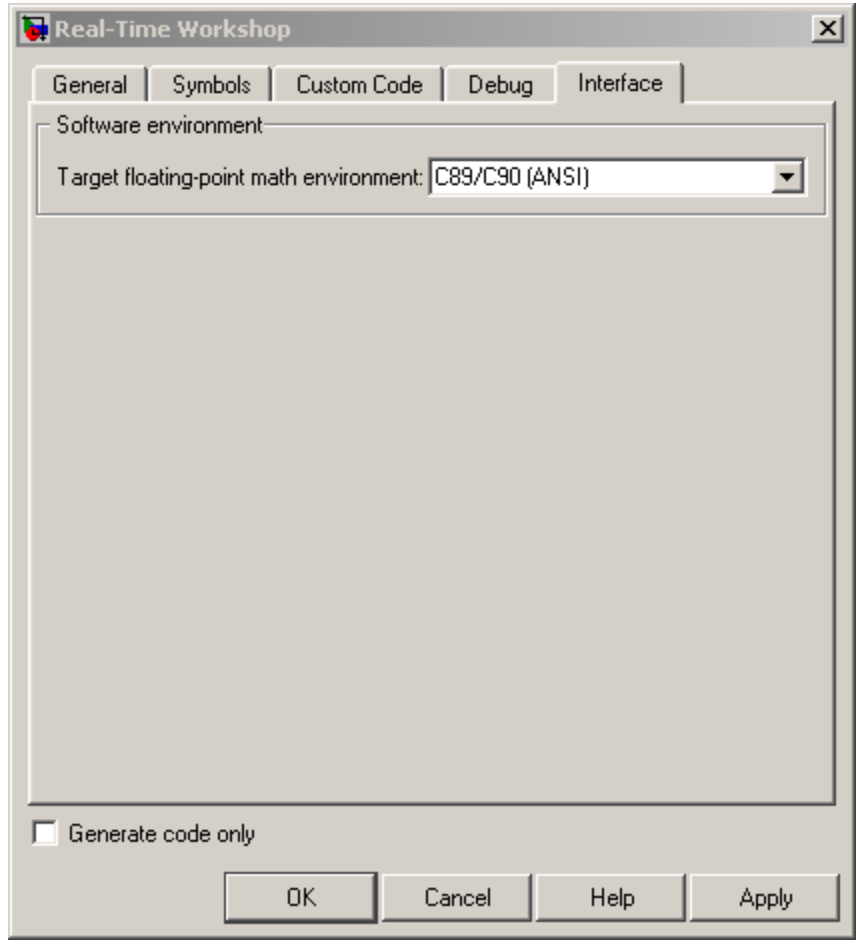
Parameters

The following table describes the debug parameters for the Embedded MATLAB Coder Real-Time Workshop dialog box:

Debug Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Verbose build" on page 6-97	`RTWVerbose` `true`, **`false`**	Display code generation progress.

Interface Tab

Specifies parameters for selecting the target software environment for the code generated by Embedded MATLAB Coder.



Parameters

The following table describes the interface parameters for the Embedded MATLAB Coder Real-Time Workshop dialog box:

Interface Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Target function library" on page 6-106	`GenFloatMathFcnCalls` *`string`*, `'`**`ANSI_C`**`'`	Specify floating-point math library extension.

Generate code only

Specify code generation versus an executable build. See "Generate code only" on page 6-31.

Automatic C MEX Generation Dialog Box for Embedded MATLAB Coder

In this section...

Automatic C MEX Generation Dialog Box Overview

Specifies parameters for C MEX generation using Embedded MATLAB Coder.

Displaying the Dialog Box

To display the Automatic C MEX Generation dialog box for Embedded MATLAB Coder, follow these steps at the MATLAB command prompt:

1. Define a configuration object variable for C MEX generation in the MATLAB workspace by issuing a constructor command like this:

   ```
   mexgen_cfg=emlcoder.MEXConfig;
   ```

2. Open the property dialog box using one of these methods:

   - Double-click the configuration object variable in the MATLAB workspace
   - Issue the `open` command from the MATLAB prompt, passing it the configuration object variable, as in this example:

     ```
     open mexgen_cfg;
     ```

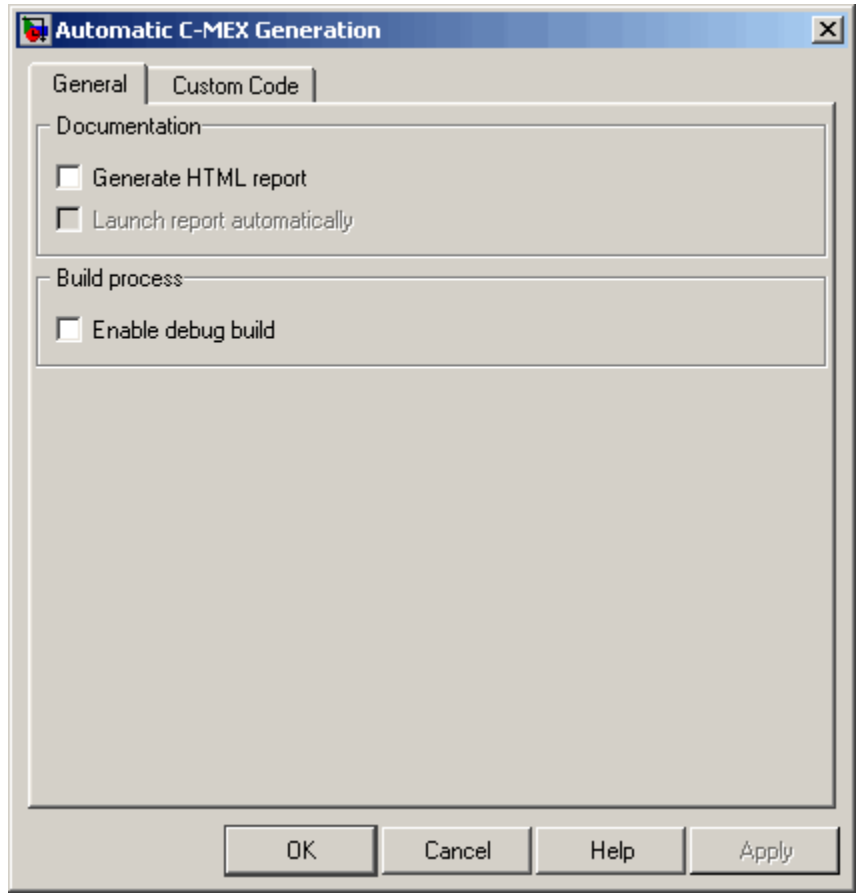
   The dialog box displays on your desktop.

See Also

"Configuring Your Environment for Code Generation"

General Tab

Specifies general parameters for C MEX generation using Embedded MATLAB Coder.



Parameters

The following table describes the general parameters for the Embedded MATLAB Coder Automatic C MEX Generation dialog box:

General Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Generate HTML report" on page 6-9	`GenerateReport` `true`, **`false`**	Document generated code in an HTML report.
"Launch report automatically" on page 6-11	`LaunchReport` `true`, **`false`**	Specify whether to automatically display HTML reports after code generation completes. **Note** Requires that you select **Generate HTML report**
"Enable debug build" on page 7-16	`EnableDebugging` `true`, **`false`**	Compile the generated code in debug mode.

Enable debug build

For C MEX code generation, specify whether Embedded MATLAB Coder compiles the generated code in debug mode.

Settings. Default: `off`

 On

Compile generated code in debug mode.

 Off

Compile generated code in release (or optimized) mode.

Command-Line Information.

Parameter: `EnableDebugging`
Type: string

Value: `'on' | 'off'`
Default: `'off'`

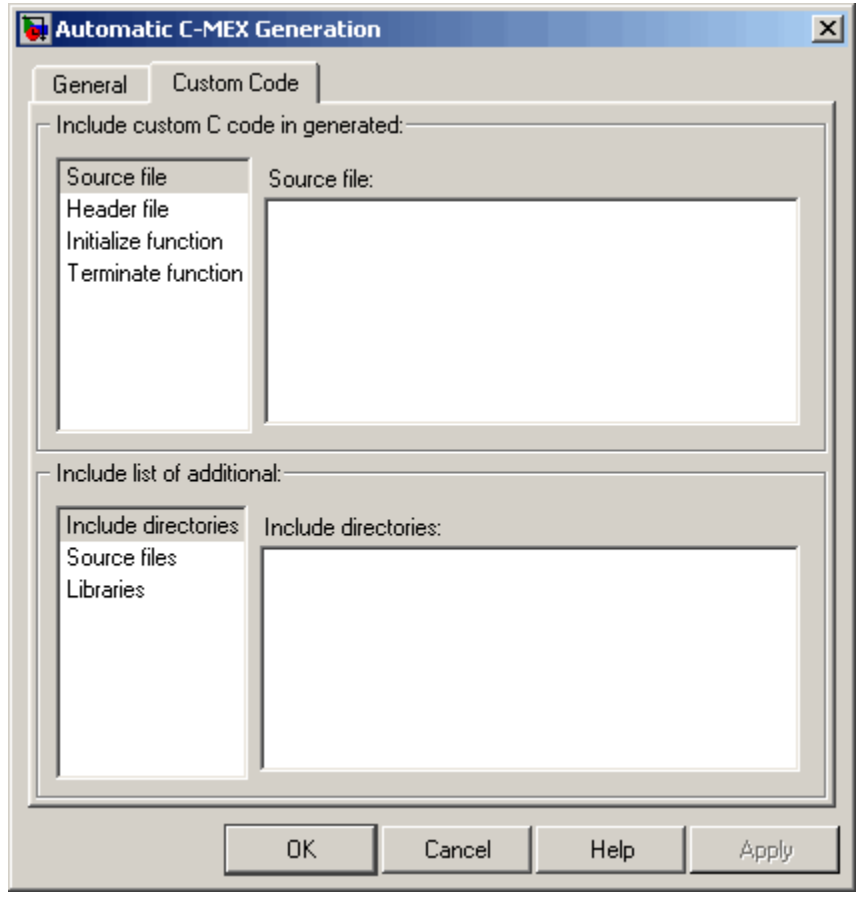
Recommended Settings.

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

See Also. "How Debugging Affects Simulation Speed" in the Simulink User's Guide documentation.

Custom Code Tab

Creates a list of custom C code, directories, source and header files, and libraries to be included in files generated by Embedded MATLAB Coder.



Configuration

1. Select the type of information to include from the list on the left side of the pane.
2. Enter a string to identify the specific code, directory, source file, or library.
3. Click **Apply**.

Parameters

The following table describes the custom code parameters for the Embedded MATLAB Coder Automatic C MEX Generation dialog box:

Custom Code Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Source file" on page 6-87	`CustomSourceCode` *`string`*, `''`	Specify code appearing near the top of the generated C MEX file.
"Header file" on page 6-88	`CustomHeaderCode` *`string`*, `''`	Specify code appearing near the top of the generated header `.h` file.
"Initialize function" on page 6-89	`CustomInitializer` *`string`*, `''`	Specify code appearing in the initialize function of the generated C MEX file.
"Terminate function" on page 6-90	`CustomTerminator` *`string`*, `''`	Specify code appearing in the terminate function of the generated `.c` or `.cpp` file.
"Include directories" on page 6-91	`CustomInclude` *`string`*, `''`	Specify a space-separated list of include directories to be added to the include path when compiling the generated code.

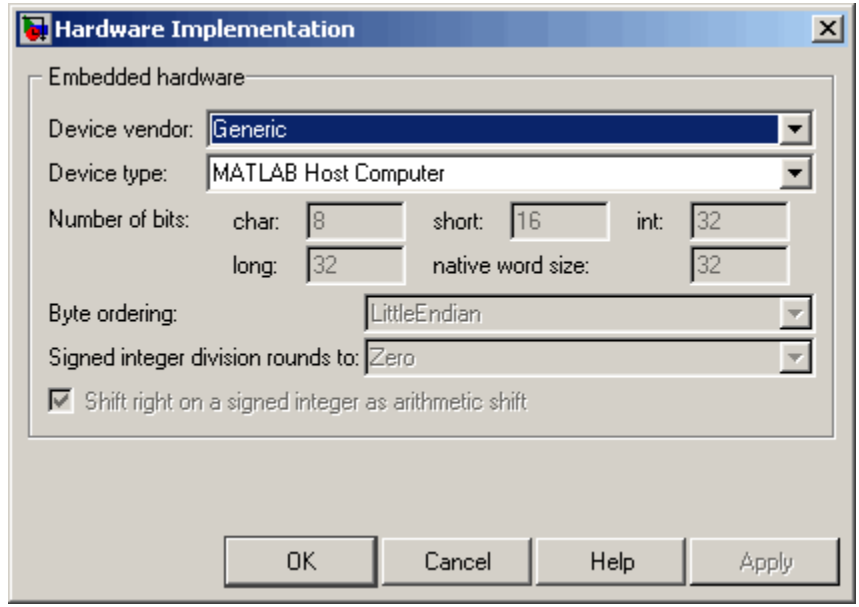
Custom Code Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Source files" on page 6-92	`CustomSource` *`string`*, `''`	Specify a space-separated list of source files to be compiled and linked with the generated code.
"Libraries" on page 6-93	`CustomLibrary` *`string`*, `''`	Specify a list of additional libraries to link with.

Hardware Implementation Dialog Box for Embedded MATLAB Coder

In this section...

Hardware Implementation Parameters Dialog Box Overview

Specifies parameters of the target hardware implementation.



Displaying the Dialog Box

To display the Hardware Implementation dialog box dialog box for Embedded MATLAB Coder, follow these steps at the MATLAB command prompt:

1 Define a configuration object variable for hardware implementation in the MATLAB workspace by issuing a constructor command like this:

```
hwi_cfg=emlcoder.HardwareImplementation;
```

2 Open the property dialog box using one of these methods:

- Double-click the configuration object variable in the MATLAB workspace
- Issue the `open` command from the MATLAB prompt, passing it the configuration object variable, as in this example:

```
open hwi_cfg;
```

The dialog box displays on your desktop.

See Also

"Configuring Your Environment for Code Generation"

Hardware Implementation Parameters

The following table describes the hardware implementation parameters for Embedded MATLAB Coder:

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Device vendor"	`ProdHWDeviceType` *`string`*, `'`**`Generic->MATLAB Host Computer`**`'`	Specify manufacturer of hardware you will use to implement the production version of the system.
"Device type"	`ProdHWDeviceType` *`string`*, `'`**`Generic->MATLAB Host Computer`**`'`	Specify type of hardware you will use to implement the production version of the system.
"Number of bits: char"	`ProdBitPerChar` *`integer`*, **`8`**	Specify length in bits of the C `char` data type supported by the target hardware.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
"Number of bits: short"	`ProdBitPerShort` *integer*, **16**	Specify length in bits of the C `short` data type supported by the target hardware.
"Number of bits: int"	`ProdBitPerInt` *integer*, **32**	Specify length in bits of the C `int` data type supported by the target hardware.
"Number of bits: long"	`ProdBitPerLong` *integer*, **32**	Specify length in bits of the C `long` data type supported by the target hardware.
"Number of bits: native word size"	`WordSize` *integer*, **32**	Specify microprocessor native word size for the target hardware.
"Byte ordering"	`ProdEndianess` `'Unspecified'`, **`'LittleEndian'`**, `'BigEndian'`	Specify significance of the first byte of a data word for the target hardware.
"Signed integer division rounds to:"	`ProdIntDivRoundTo` `'Undefined'`, **`'Zero'`**, `'Floor'`	Specify how your compiler rounds the result of dividing one signed integer by another to produce a signed integer quotient.
"Shift right on a signed integer as arithmetic shift"	`ProdShiftRightIntArith` **`true`**, `false`	Specify whether your compiler implements a signed integer right shift as an arithmetic right shift.

8

Model Advisor Checks

Real-Time Workshop Checks

In this section...

Real-Time Workshop Overview

Use Real-Time Workshop Model Advisor checks to configure your model for code generation.

See Also

- Consulting Model Advisor
- Simulink Model Advisor Check Reference
- Simulink Verification and Validation Model Advisor Check Reference

Check solver for code generation

Check model solver and sample time configuration settings.

Description

Incorrect configuration settings can stop Real-Time Workshop from generating code. Underspecifying sample times can lead to undesired results. Avoid generating code that might corrupt data or produce unpredictable behavior.

Results and Recommended Actions

Condition	Recommended Action
The solver type is set incorrectly for model level code generation.	Set **Configuration Parameters** > **Solver** > • **Type** to `Fixed-step` • **Solver** to `discrete (no continuous states)`
Multitasking diagnostic options are not set to `error`.	Set **Configuration Parameters** > **Diagnostics** > • **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. Real-Time Workshop Embedded Coder automatically changes **Solver type** to `fixed-step` when you select **Real-Time Workshop** > **Build Subsystem** or **Real-Time Workshop** > **Generate S-Function** from the subsystem context menu.

See Also

- “Adjusting Simulation Configuration Parameters for Code Generation”
- “Executing Multitasking Models”

Identify questionable blocks within the specified system

Identify blocks not supported by code generation or not recommended for deployment.

Description

Real-Time Workshop generates 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 Real-Time Workshop.	Remove the specified block from the model or replace the block with the recommended block.
A block is not recommended for production code deployment.	Remove the specified block from the model or replace the block with the recommended block.
Check for Gain blocks whose value equals 1.	Replace Gain blocks with Signal Conversion blocks.

See Also

"Requirements and Restrictions for ERT-Based Simulink Models"

Check for model reference configuration mismatch

Identify referenced model configuration parameter settings that do not match the top-level model configuration parameter settings.

Description

Real-Time Workshop cannot generate code for top-level models that contain referenced models with different, incompatible configuration parameter settings.

Results and Recommended Actions

Condition	Recommended Action
The top-level model and the referenced model have inconsistent configuration parameter settings.	Modify the specified Configuration Parameters settings.

See Also

Model Referencing Configuration Parameter Requirements

Check the hardware implementation

Identify inconsistent or underspecified hardware implementation settings

Description

Simulink and Real-Time Workshop require two sets of target specifications. The first set describes the final intended production target. The second set describes the currently selected target. If the configurations do not match, Real-Time Workshop generates extra code to emulate the behavior of the production target. Inconsistencies or underspecification of hardware attributes can lead to nonoptimal results.

Results and Recommended Actions

Condition	Recommended Action
Your system target file is `grt.tlc`.	Use an ERT-based target to generate final production code.
Hardware implementation parameters are not set to recommended values.	Specify the following **Configuration Parameters** > **Hardware Implementation** parameters to the recommended values: • **Number of bits** • **Byte ordering** • **Signed integer division rounding**
Hardware implementation **Embedded Hardware** settings do not match **Emulation Hardware** settings.	Select the **Configuration Parameters** > **Hardware Implementation** > **None** check box and configure the **Emulation hardware** settings.

Limitations

A Real-Time Workshop Embedded Coder license is required to use an ERT-based target.

See Also

Making GRT-Based Targets ERT-Compatible

Identify questionable software environment specifications

Identify questionable software environment settings.

Description

- Support for some software environment settings can lead to inefficient code generation and nonoptimal results.
- Industry standards for C, such as ISO® and MISRA®, require identifiers to be unique within the first 31 characters.
- Stateflow charts with weak Simulink I/O data types lead to inefficient code.

Results and Recommended Actions

Condition	Recommended Action
The maximum identifier length does not conform with industry standards for C.	Set the **Configuration Parameters** > **Real-Time Workshop** > **Interface** > **Maximum identifier length** parameter to 31 characters.
Real-Time Workshop Interface parameters are not set to recommended values.	Set the following **Configuration Parameters** > **Real-Time Workshop** > **Interface** parameters to the recommended values: • **Support: continuous time** • **Support: non-finite numbers** • **Support: non-inlined S-functions** • **Generate scalar inlined parameters**

Condition	Recommended Action
Real-Time Workshop Symbols parameters are not set to recommended values.	Set the **Configuration Parameters** > **Real-Time Workshop** > **Symbols** > **Generate scalar inlined parameters as** parameter to `Macros`.
The model contains Stateflow charts with weak Simulink I/O data type specifications.	Select the Stateflow chart property **Use Strong Data Typing with Simulink I/O**. You might need to adjust the data types in your model after selecting the property.

Limitations

A Stateflow license is required when using Stateflow charts.

See Also

“Strong Data Typing with Simulink I/O”

Identify questionable code instrumentation (data I/O)

Identify questionable code instrumentation.

Description

- Instrumentation of the generated code can cause nonoptimal results.
- Test points require global memory and are not optimal for production code generation.

Results and Recommended Actions

Condition	Recommended Action
Interface parameters are not set to recommended values.	Set the **Configuration Parameters > Real-Time Workshop > Interface** parameters to the recommended values.
Blocks generate assertion code.	Set the **Configuration Parameters > Diagnostics > Data Validity > Model Verification block enabling** parameter to `Disable All` on a block-by-block basis or globally.
Block output signals have one or more test points.	Remove test points from the specified block output signals. For each signal, in the Signal Properties dialog box, clear the **Test point** check box.

Check for blocks that have constraints on tunable parameters

Identify blocks with constraints on tunable parameters.

Description

Lookup Table and Lookup Table (2-D) blocks have strict constraints when they are tunable. If you violate lookup table block restrictions, the generated code produces wrong answers.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: • Preserve monotonicity of the setting for the **Vector of input values** parameter. • Preserve the number and location of zero values that you specify for **Vector of input values** and **Vector of output values** parameters if you specify multiple zero values for the **Vector of input values** parameter.
Lookup Table (2-D) blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: • Preserve monotonicity of the setting for the **Row index input values** and **Column index of input values** parameters. • Preserve the number and location of zero values that you specify for **Row index input values**, **Column index of input values**, and **Vector of output values** parameters if you specify multiple zero values for the **Row index input values** or **Column index of input values** parameters.

See Also

Lookup Table block

Identify questionable subsystem settings

Identify questionable subsystem block settings.

Description

Subsystem blocks implemented as void/void functions in the generated code use global memory to store the subsystem I/O.

Results and Recommended Actions

Condition	Recommended Action
Subsystem blocks have the **Subsystem Parameters** > **Real-Time Workshop system code** option set to `Function`.	Set the **Subsystem Parameters** > **Real-Time Workshop system code** parameter to `Auto`.

See Also

Subsystem block

Identify blocks that generate expensive saturation and rounding code

Check for blocks that generate expensive saturation or rounding code.

Description

- Setting the **Saturate on integer overflow** parameter can produce condition-checking code that your application might not require.
- Generated rounding code is inefficient because of **Round integer calculations toward** parameter setting.

Results and Recommended Actions

Condition	Recommended Action
Blocks generate expensive saturation code.	Check each block to ensure that your application requires setting **Function Block Parameters** > **Signal Attributes** > **Saturate on integer overflow**. Otherwise, clear the **Saturate on integer overflow** parameter to ensure the most efficient implementation of the block in the generated code.
Generated code is inefficient.	Set the **Function Block Parameters** > **Round integer calculations toward** parameter to the recommended value.

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.

Results and Recommended Actions

Condition	Recommended Action
The model represents a multirate system but is not configured for multitasking.	Set the **Configuration Parameters > Solver > Tasking mode for periodic sample times** parameter as recommended.
The model is configured for multitasking, but multitasking is not appropriate for the target hardware.	Set the **Configuration Parameters > Solver > Tasking mode for periodic sample times** parameter to `SingleTasking`, or change the **Configuration Parameters > Hardware Implementation** settings.

See Also

"Single-Tasking and Multitasking Execution Modes"

Identify questionable fixed-point operations

Identify fixed-point operations that can lead to nonoptimal results.

Description

The following operations can lead to nonoptimal results:

- Division
  - The rounding behavior of signed integer division is not fully specified by C language standards. Therefore, the generated code for division is large to ensure bit-true agreement between simulation and code generation.
  - Integer division generated code contains protection against arithmetic exceptions such as division by zero, INT_MIN/-1, and LONG_MIN/-1. If you construct models making it impossible for exception triggering input combinations to reach a division operation, the protection code generated as part of the division operation is redundant.
  - The index search method `Evenly-spaced points` requires a division operation, which can be computationally expensive.
- Multiplication
  - Product blocks are configured to do more than one division operation. Multiplying all the denominator terms together first, and then computing only one division operation improves accuracy and speed in floating-point and fixed-point calculations.
  - Product blocks are configured to do more than one multiplication or division operation. Using several blocks, with each block performing one multiplication or one division operation, allows you to control the data type and scaling used for intermediate calculations. The choice of data types for intermediate calculations affects precision, range errors, and efficiency.
  - Blocks that have the **Saturate on integer overflow** parameter selected, and have an ideal multiplication product with a larger integer size than the target integer size, must determine the ideal product in generated C code. The C code required to do this multiplication is large and slow.

  - Blocks with relative scaling of inputs and outputs must determine the ideal product in the generated C code. The C code required to do this multiplication is large and slow.
  - Blocks that multiply signals with nonzero bias require extra steps to implement the multiplication. Inserting Data Type Conversion blocks remove the biases, and allow you to control data type and scaling for intermediate calculations. The conversion is done once and all blocks in the subsystem benefit from simpler, bias-free math.
  - Blocks are multiplying signals with mismatched fractional slopes. This mismatch causes the overall operation to involve two multiply instructions.
  - Real-Time Workshop generates a reciprocal operation followed by a multiply operation for Product blocks that have a divide operation for the first input, and a multiply operation for the second input. If you reverse the inputs so that the multiplication occurs first and the division occurs second, Real-Time Workshop generates a single division operation for both inputs.
  - An input with an invariant constant value is used as the denominator in an online division operation. If the operation is changed to multiplication, and the invariant input is replaced by its reciprocal, then the division is done offline and the online operation is multiplication. This leads to faster and smaller generated code.

- Addition
  - For better accuracy and efficiency, nonzero bias terms are handled separately and are not included in the conversion from input to output. The ranges given for the input and output exclude their biases.
  - Sum blocks can have a range error before an addition or subtraction operation. For simplicity of design, the Sum block always casts each input to the output data type and scaling before performing addition or subtraction. The input range is different than the output range, so a range error can occur when casting the input to the output data type.
  - A Sum block has an input with a fractional slope that does not equal the fractional slope of the output. This mismatch requires the Sum block to multiply the input by the net slope adjustment each time the input is converted to the output data type and scaling.

  - The net sum of the Sum block input biases does not equal the bias of the output. The generated code includes one extra addition or subtraction instruction to correctly account for the net bias adjustment.

- Using Relational Operator blocks
  - The data types of the Relational Operator block inputs are not the same. A conversion operation is required every time the block is executed. If one of the inputs is invariant, then changing the data type and scaling of the invariant input to match the other input improves the efficiency of the model.
  - The Relational Operator block inputs have different ranges, resulting in a range error when casting, and a precision loss each time a conversion is performed. You can insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type that has sufficient range and precision to represent each input, making the relational operation error-free.
  - The inputs of the Relational Operator block have different fractional slopes. The mismatch causes the Relational Operator block to require a multiply operation each time the input with lesser positive range is converted to the data type and scaling of the input with greater positive range.
- Using MinMax blocks
  - The input and output of the MinMax block have different data types. A conversion operation is required every time the block is executed. The model is more efficient with the same data types.
  - The input of the MinMax block is converted to the data type and scaling of the output before performing a relational operation, resulting in a range error when casting, or a precision loss each time a conversion is performed.
  - The input of the MinMax block has a different fractional slope than the output. This mismatch causes the MinMax block to require a multiply operation each time the input is converted to the data type and scaling of the output.
- Discrete-Time Integerator blocks have a complicated initial condition setting. The initial condition for the Discrete-Time Integrator blocks are

used to initialize the state and output. As a result, the output equation generates excessive code and an extra global variable is required.

Results and Recommended Actions

Condition	Recommended Action
Integer division generated code is large.	Set the **Configuration Parameters > Hardware Implementation > Signed integer division rounds to** parameter to the recommended value.
Protection code generated as part of the division operation is redundant.	Verify that your model cannot cause exceptions in division operations and then remove redundant protection code by setting the **Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions** parameter.
Generated code is inefficient.	Set the **Function Block Parameters > Round integer calculations toward** parameter to the recommended value.
Lookup table input data is not evenly spaced.	If the data is nontunable, adjust the table to be evenly spaced. See `fixpt_look1_func_approx`.
Lookup table input data is not evenly spaced when quantized, but it is very close to being evenly spaced.	If the data is nontunable, adjust the table to be evenly spaced. See `fixpt_evenspace_cleanup`.
Lookup table input data is evenly spaced, but the spacing is not a power of 2.	If the data is nontunable, reimplement the table with even power-of-2 spacing. See `fixpt_look1_func_approx`.

Condition	Recommended Action
Index search method is set to `Evenly-spaced points`.	Specify a different **Function Block Parameters** > **Index search method** to avoid the division operation.
Blocks require cumbersome multiplication.	Restrict multiplication operations: • So the product integer size is no larger than the target integer size. • To the recommended size.
Blocks multiply signals with nonzero bias.	Insert a Data Type Conversion block before and after the block containing the multiplication operation.
Product blocks are multiplying signals with mismatched fractional slopes.	Change the scaling of the output so that its fractional slope is the product of the input fractional slopes.
Product blocks are configured to do multiple division operations.	Multiply all the denominator terms together, and then do a single division using cascading Product blocks.
Product blocks are configured to do many multiplication or division operations.	Split the operations across several blocks, with each block performing one multiplication or one division operation.
Product blocks are configured with a divide operation for the first input and a multiply operation for the second input.	Reverse the inputs so the multiply operation occurs first and the division operation occurs second.
An input with an invariant constant value is used as the denominator in an online division operation.	Change the operation to multiplication, and replace the invariant input by its reciprocal.
A Sum block has a different input and output data type range.	Insert a Data Type Conversion block before and after the Sum block.

Condition	Recommended Action
A Sum block has an input with a fractional slope that does not equal the fractional slope of the output.	Change the scaling of the input or output.
The net sum of the Sum block input biases does not equal the bias of the output.	Change the bias of the output, making the net bias adjustment zero.
The inputs of the Relational Operator block have different data types.	• Change the data type and scaling of the invariant input to match other inputs. • Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type.
The inputs of the Relational Operator block have different fractional slopes.	Change the scaling of either input.
The input and output of the MinMax block have different data types.	Change the data type of the input or output.
The input of the MinMax block has a different fractional slope than the output.	Change the scaling of the input or the output.
The initial condition of the Discrete-Time Integrator block is used to initialize both the state and the output.	Set the **Function Block Parameters** > **Use initial condition as initial and reset value for** parameter to `State only (most efficient)`.

Limitations

A Simulink Fixed Point license is required to generate fixed-point code.

See Also

- Lookup Table block

- Remove code that protects against division arithmetic exceptions

Index

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U