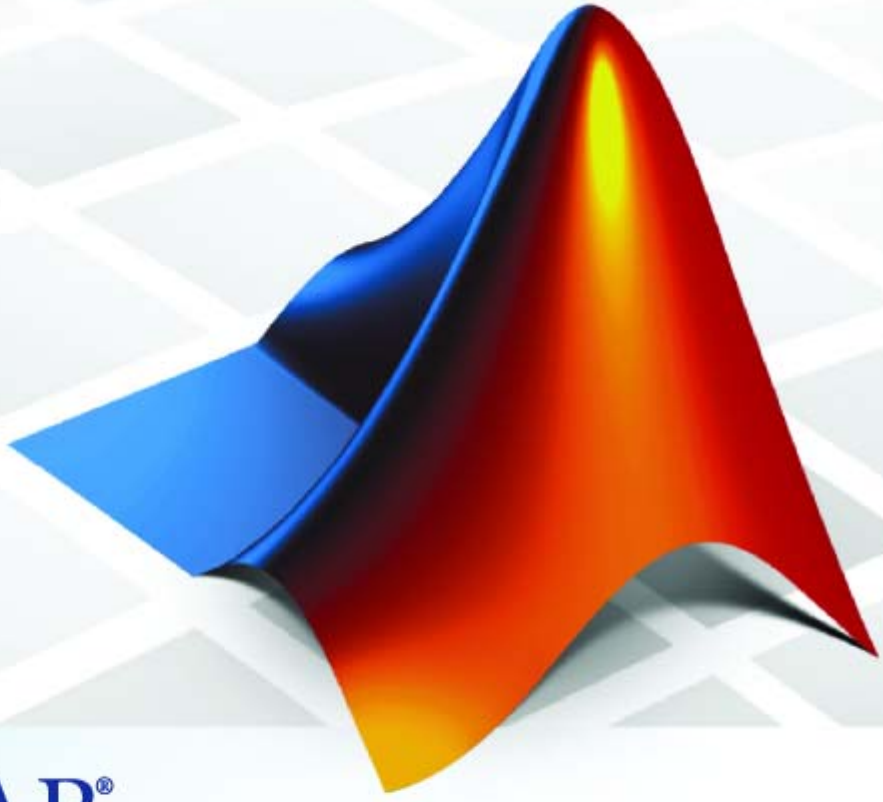
Real-Time Workshop® 6

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	Updated for Version 6.5 (Release 2006b)
March 2007	Online only	Updated for Version 6.6 (Release 2007a)

Contents

5 Blocks — Alphabetical List

6 Configuration Parameter Reference

7 Configuration Parameters Dialog Box Reference

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

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.

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"

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"

addIncludeFiles

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"

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"

addLinkFlags

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"

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.

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 (optional)
: 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`* and *`linkobjs`* arguments, you can specify any combination of the optional arguments *`paths`*, *`priority`*, *`precompiled`*, *`linkable`*, 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 *`linkable`* as a value or vector of values.

If You Specify *`priority`*, *`precompiled`*, or *`linkable`* 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`*.

For any optional argument you choose to omit between *linkobjs* and any other argument, specify a null string (''). For example, to specify that all objects are precompiled, without specifying paths or priorities, you might call addLinkObjects as

```
addLinkObjects(myBuildInfo, {'test1' test2' 'test3'},...
 '', '', 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"

addSourceFiles

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"

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"

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 "Application Lifespan". 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 "Application Lifespan". 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.

6

Configuration Parameter Reference

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 "Adjusting Simulation Configuration Parameters for Code Generation" and "Configuring Real-Time Workshop Code Generation Parameters".

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 "Model Configuration Dialog" 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
`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.
`CodeGenDirectory`	Not available	For MathWorks use only.
`CombineOutputUpdateFcns` (ERT) `off`, **`on`**	**Real-Time Workshop > Interface > Single output/update function**	Generate a model's output and update routines into a single-step function.
`Comment`	Not available	For MathWorks use only.
`ConfigAtBuild`	Not available	For MathWorks use only.
`ConfigurationMode`	Not available	For MathWorks use only.
`ConfigurationScript`	Not available	For MathWorks use only.
`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.
`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.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`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 $A - Data type acronym
`CustomSymbolStrFcn` (ERT) *`string`* - **`$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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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
`CustomTerminator` *`string`*	**Real-Time Workshop > Custom Code > Terminate function**	Specify code to appear in the model's generated terminate function.
`DataBitsets` (Stateflow) **`off`**, `on`	**Optimization > Use bit sets for storing boolean data**	Use bit sets for storing Boolean data.
`DataDefinitionFile` (ERT) *`string`*	**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`*	**Real-Time Workshop > Data Placement > Data declaration filename**	Specify the name of a single separate `.c` or `.cpp` file that contains global data references.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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) **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop** > **Symbols** > **#define naming**	Specify the rule that changes the spelling of all `#define` names.
`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.
`ERTCustomFileBanners`	Not available	For MathWorks use only.
`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.
`EnableCustomComments` (ERT) **`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.
`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.
ERTFirstTimeCompliant (ERT) `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.
`EvaledLifeSpan`	Not available	For MathWorks use only.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`ExtMode` **`off`**, `on`	**Real-Time Workshop > Interface**	Specify the data interface to be generated with the code.
`ExtModeMexArgs` *`string`* - **`mex`**	**Real-Time Workshop** > **Interface** > **Interface** > **External** > **MEX-file arguments**	Specify external mode `mex` arguments.
`ExtModeMexFile`	Not available	For MathWorks use only.
`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` **`off`**, `on`	**Real-Time Workshop** > **Interface** > **Static memory buffer size**	Specify the size in bytes of the external mode static memory buffer.
`ExtModeTesting`	Not available	For MathWorks use only.
`ExtModeTransport` **`tcpip`**, `serial-win32`	**Real-Time Workshop** > **Interface** > **Interface** > **External** > **Transport layer**	Specify transport protocols for external mode communications.
`FoldNonRolledExpr`	Not available	For MathWorks use only.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ForceParamTrailComments` **`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.
`GenCodeOnly` **`off`**, `on`	**Real-Time Workshop > Generate code only**	Generate source code, but do not execute the makefile to build an executable.
`GenerateASAP2` **`off`**, `on`	**Real-Time Workshop** > **Interface** > **Interface**	Specify the data interface to be generated with the code.
`GenerateComments` `off`, **`on`**	**Real-Time Workshop** > **Comments** > **Include comments**	Include comments in generated code.
`GenerateErtSFunction` (ERT) **`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.
`GenerateFullHeader`	Not available	For MathWorks use only.
`GenerateMakefile` `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` **`off`**, `on`	**Real-Time Workshop** > **General** > **Generate HTML report**	Document the generated C or C++ code in an HTML report.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateSampleERTMain` (ERT) **`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`.
`GenFloatMathFcnCalls` *`string`* - **`ANSI_C`**	**Real-Time Workshop** > **Interface** > **Target floating-point math environment**	Specify the math library extension available to your target. 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 `ISO_C` - ISO/IEC 9899:1999 C standard math library `GNU` - GNU gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`
`GlobalDataDefinition`(ERT) **`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) **`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
`GRTInterface` (ERT) **`off`**, `on`	**Real-Time Workshop > Interface > GRT compatible call interface**	Include a code interface (wrapper) that is compatible with the GRT target.
`IgnoreCustomStorageClasses` (ERT) `off`, **`on`**	**Real-Time Workshop > General > Ignore custom storage classes**	Treat custom storage classes as `'Auto'`.
`IncAutoGenComments`	Not available	For MathWorks use only.
`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'`.
`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`.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IncludeERTFirstTime` (ERT) **`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.
`IncludeFileDelimiter` (ERT) **`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.
`IncludeHyperlinkInReport` (ERT) **`off`**, `on`	**Real-Time Workshop** > **General** > **Include hyperlinks to model**	Link code segments to the corresponding block in the model. This option increases code generation time for large models.
`IncludeMdlTerminateFcn` (ERT) **`off`**, `on`	**Real-Time Workshop** > **Interface** >**Terminate function required**	Generate a terminate function for the model.
`IncludeRegionsInRTWFile` `BlockHierarchyMap`	Not available	For MathWorks use only.
`IncludeRootSignalInRTWFile`	Not available	For MathWorks use only.
`IncludeVirtualBlocksInRTW` `FileBlockHierarchyMap`	Not available	For MathWorks use only.

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.
`InlinedPrmAccess` (ERT) **`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.
`InsertBlockDesc` (ERT) **`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.
`IsERTTarget`	Not available	For MathWorks use only.
`IsPILTarget`	Not available	For MathWorks use only.
`LaunchReport` **`off`**, `on`	**Real-Time Workshop** > **General** > **Launch report automatically**	Display the HTML report after code generation completes.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`LogVarNameModifier` **`none`**, `rt_`, `_rt`	**Real-Time Workshop** > **Interface** > **MAT-file variable name modifier**	Augment the MAT-file variable name.
`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.
`MangleLength` `slint` - **`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.
`MatFileLogging` (ERT) **`off`**, `on`	**Real-Time Workshop** > **Interface** > **MAT-file logging**	Generate code that logs data to a MATLAB `.mat` file.
`MaxIdLength` `slint` - **`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.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MemSecPackage` (ERT) *`string`* - **`--- None ---`**	**Real-Time Workshop** > **Memory Sections** > **Package**	Specify the package that contains the memory sections that you want to apply.
`MemSecFuncInitTerm` (ERT) *`string`* - **`Default`**	**Real-Time Workshop** > **Memory Sections** > **Initialize/Terminate**	Apply memory sections to: • Initialize/Start functions • Terminate functions
`MemSecFuncExecute` (ERT) *`string`* - **`Default`**	**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`**	**Real-Time Workshop** > **Memory Sections** > **Constants**	Apply memory sections to: • Constant parameters • Constant block I/O • Zero representation
`MemSecDataIO` (ERT) *`string`* - **`Default`**	**Real-Time Workshop** > **Memory Sections** > **Inputs/Outputs**	Apply memory sections to: • Root inputs • Root outputs

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MemSecDataInternal` (ERT) *`string`* - **`Default`**	**Real-Time Workshop** > **Memory Sections** > **Internal data**	Apply memory sections to: • Block I/O • D-work vectors • Run-time model • Zero-crossings
`MemSecDataParameters` (ERT) *`string`* - **`Default`**	**Real-Time Workshop** > **Memory Sections** > **Parameters**	Apply memory sections to: • Parameters
`ModelReferenceCompliant`	Not available	Set in `SelectCallback` for a target to indicate whether the target supports model reference.
`ModelStepFunctionPrototypeControlCompliant` (ERT) `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.
`ModuleName` (ERT) *`string`*	**Real-Time Workshop** > **Placement** > **Module name**	Specify the name of the module that owns this model.
`ModuleNamingRule` (ERT) **`Unspecified`**, `SameAsModel`, `UserSpecified`	**Real-Time Workshop** > **Data Placement** > **Module naming**	Specify the rule to be used for naming the module.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MultiInstanceErrorCode` (ERT) `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) **`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.
`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.
`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.

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.
`ParamNamingFcn`	Not available	For MathWorks use only.
`ParamNamingRule` (ERT) **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop** > **Symbols** > **Parameter naming**	Select a rule that changes spelling of all parameter names.
`ParamTuneLevel` (ERT) `slint -` **`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.
`ParenthesesLevel` `minimum`, **`nominal`**, `maximum`	**Real-Time Workshop > Code Style > Parentheses Level**	Control existence of optional parentheses in generated code.
`PortableWordSizes` (ERT) **`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.
`PostCodeGenCommand` `string`	Not available	Add the specified post code generation command to the model's build process.

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.
`PreserveExpressionOrder` (ERT) **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve operand order in expression**	Control reordering of commutable expressions.
`PreserveIfCondition` (ERT) **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve condition expression in if statement**	Control preservation of `if` statement conditions.
`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.
`ProfileTLC` **`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.
`PurelyIntegerCode` (ERT) **`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.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`RTWCAPIParams` **`off`**, `on`	**Real-Time Workshop** > **Interface** > **Parameters in C API**	Generate parameter tuning structures in C API.
`RTWCAPISignals` **`off`**, `on`	**Real-Time Workshop** > **Interface** > **Signals in C API**	Generate signal structure in C API.
`RTWCAPIStates`	Not available	For MathWorks use only.
`RTWVerbose` `off`, **`on`**	**Real-Time Workshop** > **Debug** > **Verbose build**	Display messages indicating code generation stages and compiler output.
`ReqsInCode` (ERT) **`off`**, `on`	**Real-Time Workshop** > **Comments** > **Requirements in block comments**	Include specified requirements in the generated code as a comment.
`RetainRTWFile` **`off`**, `on`	**Real-Time Workshop** > **Debug** > **Retain .rtw file**	Retain the *`model`*`.rtw` file in the current build directory.
`RollThreshold` `slint -` **`5`**	**Optimization > Loop unrolling threshold**	Specify the minimum signal width for which a `for` loop is to be generated.
`RootIOFormat` (ERT) **`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.
`RSIM_STORAGE_CLASS_AUTO`	**Real-Time Workshop > RSim Target > Force storage classes to AUTO**	Force all storage classes for a model to `Auto`.
`SaveLog` **`off`**, `on`	**Real-Time Workshop > General > Save build log**	Save build log.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SFDataObjDesc` (ERT) **`off`**, `on`	**Real-Time Workshop** > **Comments** > **Stateflow object descriptions**	Insert Stateflow object descriptions into the generated code as a comment.
`ShowEliminatedStatements` **`off`**, `on`	**Real-Time Workshop** > **Comments** > **Show eliminated blocks**	Show statements for eliminated blocks as comments in the generated code.
`SignalDisplayLevel` (ERT) `slint - 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.
`SignalLabelMismatchMsg` **`None`**, `Warning`, `Error`	**Diagnostics > Connectivity > Signal label mismatch**	Specify the diagnostic action to take when a signal label mismatch occurs.
`SignalNamingFcn`	Not available	For MathWorks use only.
`SignalNamingRule` (ERT) **`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.
`SimulinkBlockComments` `off`, **`on`**	**Real-Time Workshop** > **Comments** > **Simulink block comments**	Insert Simulink block names as comments above the generated code for each block.
`SimulinkDataObjDesc` (ERT) **`off`**, `on`	**Real-Time Workshop** > **Comments** > **Simulink data object descriptions**	Insert Simulink data object descriptions into the generated code as comments.
`StateBitsets` (Stateflow) **`off`**, `on`	**Optimization > Use bit sets for storing state configuration**	Use bit sets for storing state configuration.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SupportAbsoluteTime` (ERT) **`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) **`off`**, `on`	**Real-Time Workshop** > **Interface** > **complex numbers**	Support complex data types in the generated code.
`SupportContinuousTime` (ERT) `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) **`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` `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) **`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`.
`SystemCodeInlineAuto`	Not available	For MathWorks use only.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SystemTargetFile` `string`	**Real-Time Workshop** > **General** > **System target file**	Specify a system target file.
`TargetBitPerChar` `slint - 8`	**Hardware Implementation > Emulation hardware > char**	Specify the number of bits used to represent the C/C++ type `char`.
`TargetBitPerInt` `slint - 32`	**Hardware Implementation > Emulation hardware > int**	Specify the number of bits used to represent the C/C++ type `int`.
`TargetBitPerLong` `slint - 32`	**Hardware Implementation > Emulation hardware > long**	Specify the number of bits used to represent the C/C++ type `long`.
`TargetBitPerShort` `slint - 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.
`TargetFcnLib`	Not available	For MathWorks use only.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TargetHWDeviceType` *`string`*	**Hardware Implementation > Emulation hardware > Device type**	Specify a predefined hardware device to define the C or C++ language constraints for your microprocessor or `Custom` if your microprocessor is not listed. Specify the string `"MATLAB Host Computer"` to target the current MATLAB host machine.
`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.
`TargetLang` **`C`**, `C++`	**Real-Time Workshop > Language**	Specify whether Real-Time Workshop is to generate C or C++ code.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TargetLibSuffix` *`string`*	Not available	Control the suffix used for naming a target's dependent libraries (for example, `_target.a`). An example of when you might use this is for generated model reference libraries. If you do not set this parameter, on a Windows system, you get *`modelName`*`_rtwlib.lib` and on a UNIX system, you get *`modelName`*`_rtwlib.a`.
`TargetOS` (ERT) **`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.
`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`.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TargetPreprocMaxBitsSint` `int - 128`	Not available	Specify the maximum number of bits that the target C preprocessor can use for signed integer math.
`TargetPreprocMaxBitsUint` `int - 128`	Not available	Specify the maximum number of bits that the target C preprocessor can use for unsigned integer math.
`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.
`TargetTypeEmulationWarn 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.
`TargetWordSize` `slint - 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.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TemplateMakefile` *`string`* - **`grt_default_tmf`**	**Real-Time Workshop** > **General** > **Template makefile**	Specify the current template makefile for building a Real-Time Workshop target.
`TLCAssert` **`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` **`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` **`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.
`TLCOptions` *`string`*	**Real-Time Workshop > General > TLC options**	Specify additional TLC command line options.
`UseTempVars` (Stateflow) **`off`**, `on`	**Optimization > Minimize array reads using temporary variables**	Minimize array reads in global memory by using temporary variables.
`UtilityFuncGeneration` **`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
`ZeroExternalMemoryAt Startup` (ERT) `off`, **`on`**	**Optimization > Remove root level I/O zero initialization**	Suppress code that initializes root-level I/O data structures to zero.
`ZeroInternalMemoryAt Startup` (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.

7

Configuration Parameters Dialog Box Reference

Solver

Start time

Enter a double-precision value scaled to seconds specifying simulation or generated code start time

Default: 0.0

- A start time other than 0.0 represents an offset, and must be less than or equal to the stop time. An example of when you might use an offset is to set up a delay to accommodate some type of initialization.
- The values of block parameters with initial conditions must match the initial condition settings at the specified start time.
- Simulation time is not the same as clock time. For example, running a simulation for 10 seconds usually does not take 10 seconds. Total simulation time depends on factors such as model complexity, solver step sizes, and computer speed.

Command line parameter

`StartTime`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Stop time

Enter a double-precision value scaled to seconds specifying simulation or generated code stop time

Default: 10

- Stop time must be greater than or equal to the start time.
- Specify `inf` to run a simulation or generated program until you explicitly pause or stop it.
- If the stop time is the same as the start time, the simulation or generated program runs for one step.
- Simulation time is not the same as clock time. For example, running a simulation for 10 seconds usually does not take 10 seconds. Total simulation time depends on factors such as model complexity, solver step sizes, and computer speed.
- If your model includes blocks that depend on absolute time and you are creating a design that runs indefinitely, see Blocks That Depend on Absolute Time.

Command line parameter

`StopTime`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

Type

Specify the type of solver to be applied to your model: variable-step or fixed-step

The solver computes the next time as the sum of the current time and the step size.

Variable-step (default)
Step size varies from step to step, depending on model dynamics.

- Reduces step size when model states change rapidly, to maintain accuracy.
- Increases step size when model states change slowly, to avoid unnecessary steps.

Recommended if the model's states change rapidly or contain discontinuities. It shortens simulation time significantly because it requires fewer time steps than a fixed-step solver to achieve a comparable level of accuracy.

Fixed-step
Step size remains constant throughout the simulation.

Required for code generation, unless you use an S-function or RSim target.

Dependencies

Selecting **Variable-step** enables the following options:

- **Max step size**
- **Min step size**
- **Initial step size**
- **Solver**
- **Relative tolerance**
- **Absolute tolerance**
- **Zero crossing control**
- **Number of consecutive min step size violations allowed**
- **Consecutive zero crossings relative tolerance**
- **Number of consecutive zero crossings allowed**

Selecting **Fixed-step** enables the following options:

- **Solver**
- **Periodic sample time constraint**
- **Fixed-step size (fundamental sample time)**
- **Tasking mode for periodic sample times**
- **Higher priority value indicates higher task priority**
- **Automatically handle data transfers between tasks**

Command line parameter

`SolverType`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- Choosing a Solver Type
- Determining Step Size for Discrete Systems

Tasking mode for periodic sample times

Indicate how blocks with periodic sample times are to execute

Auto (default)
: Single-tasking execution is used if:

  - Your model contains one sample time.
  - Your model contains a continuous and a discrete sample time, and the fixed-step size is equal to the discrete sample time.

Selects multitasking execution for a models operating at different sample rates.

SingleTasking
Process all blocks through each stage of simulation (for example, calculating output and updating discrete states) together. For more information, see Single-Tasking Mode.

MultiTasking
Process groups of blocks with the same execution priority through each stage of simulation (for example, calculating output and updating discrete states) based on task priority. Multitasking mode helps to create valid models of real-world multitasking systems, where sections of your model represent concurrent tasks. For more information, see Multitasking and Pseudomultitasking Modes.

The **Multitask rate transition** option on the **Diagnostics > Sample Time** pane allows you to adjust error checking for sample rate transitions between blocks that operate at different sample rates.

Dependency

Enabled by selecting **Fixed-step** solver type.

Command line parameter

`SolverMode`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- Rate Transition block

- Model Execution and Rate Transitions
- Single-Tasking Versus Multitasking Operation
- Sample Rate Transitions
- Single-Tasking and Multitasking Execution of a Model: an Example

Optimization

Block reduction

Reduce execution time by collapsing or removing groups of blocks

Checked(default)

Simulink searches for and reduces the following block patterns:

- Accumulators—pattern consisting of a constant block, a Sum block, and feedback through a Unit Delay block
- Redundant type conversions—for example, an `int` type conversion block with an input and output of type `int`
- Dead code—blocks or signals in an unused code path
- Fast-to-slow Rate Transition block in a single-tasking system—the Rate Transition block's input frequency is faster than its output frequency

Unchecked

Simulink does not search for instances of block patterns for block reduction optimization. Simulation and generated code are not optimized.

Tips

- Block reduction is only intended to remove the code that represents execution of a block. Other supporting data, such as definitions for sample time and data types might remain in the generated code.
- Tunable parameters do not prevent a block from being reduced by dead code elimination.

Command line parameter

`BlockReduction`

Recommended settings

Debugging	Clear
Traceability	Clear
Efficiency	Set
Safety precaution	No impact

More information

- “Block Reduction”
- “Single-Tasking Execution”

Conditional input branch execution

Improve model execution when the model contains Switch and Multiport Switch blocks

Checked (default)
Only the blocks required to compute the control input and the data input selected by the control input are executed. This optimization speeds execution of code generated from the model. Limits to Switch block optimization:

- Only blocks with `-1` (inherited) or `inf` (Constant) sample time can participate.
- Blocks with outputs flagged as test points cannot participate.
- No multirate block can participate.
- Blocks with states cannot participate.
- Only S-functions with option `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` set can participate.

Unchecked
Executes all blocks driving the Switch block input ports at each time step.

Command line parameter

`ConditionallyExecuteInputs`

Recommended settings

Debugging	No impact
Traceability	Set
Efficiency	Set
Safety precaution	No impact

More information

"Conditional Input Execution"

Implement logic signals as boolean data (vs. double)

Enables error detection for mixed double/Boolean types.

Checked (default)
: Enables Boolean type checking, resulting in an error when double signals are connected to blocks that prefer Boolean inputs. Generated code requires less memory with this enabled.

Unchecked
: Does not produce an error when double signals are connected to blocks that prefer Boolean inputs. This ensures compatibility with models created by earlier versions of Simulink that support only double data types.

Dependency

Disable for models created with a version of Simulink that supports only signals of type `double`.

Command line parameter

`BooleanDataType`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	No impact

More information

"Implement Logic Signals as Boolean Data"

Signal storage reuse

Reuse signal memory. Only applies to signals with storage class `Auto`.

Checked (default)
: Instructs Real-Time Workshop to reuse signal memory, reducing the memory requirement of your real-time program.

Unchecked

Makes all block outputs global and unique, which in many cases significantly increases RAM and ROM usage.

Dependencies

Enables the following options

- **Enable local block outputs**
- **Reuse block outputs**
- **Eliminate superfluous temporary variables (Expression folding)**

Command line parameter

`OptimizeBlockIOStorage`

Recommended settings

Debugging	Clear
Traceability	Clear
Efficiency	Set
Safety precaution	No impact

More information

- “Signal Storage, Optimization, and Interfacing”
- “Signal Storage Concepts”

Inline parameters

Transform tunable parameters into constant values

Checked

Enabling Inline parameters has three effects:

- Real-Time Workshop uses the numerical values of model parameters, instead of their symbolic names, in generated code.

- Reduces global RAM usage, because parameters are not declared in the global parameters structure.
- The **Configure** button becomes enabled. Clicking the **Configure** button opens the Model Parameter Configuration dialog box.

Unchecked (default)
Uses model parameters symbolic names in generated code.

Tips

When a top-level model uses referenced models:

- All referenced models must specify **Inline parameters** to be `on`.
- The top-level model can specify **Inline parameters** to be `on` or `off`.

Dependencies

Disable for referenced models in a model reference hierarchy.

Enables the following options:

- **Configure** button
- **Inline invariant signals**

Command line parameter

`InlineParams`

Recommended settings

Debugging	Clear
Traceability	Set
Efficiency	Set
Safety precaution	No impact

More information

- “Parameter Storage, Interfacing, and Tuning”
- “Inline Parameters”

Application lifespan (days)

Optimize the size of counters used to compute absolute and elapsed time

Default: `inf`

Min: `8 bits`

Max: `inf`

Tips

- A timer will allocate 64 bits of memory for a timer if you specify a value of **inf**.
- To minimize the amount of RAM used by time counters, specify a lifespan no longer than necessary.
- Must be the same for top and referenced models.

Command line parameter

`LifeSpan`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Finite value
Safety precaution	`inf`

More information

- “Application Lifespan”
- “Using Timers in Asynchronous Tasks”

Enable local block outputs

Specify whether block signals are declared locally or globally

Checked (default)
: Block signals are declared locally in functions.

Unchecked
: Block signals are declared globally.

Tips

- If it is not possible to declare an output as a local variable, the generated code declares the output as a global variable.
- If you are constrained by limited stack space, you can turn **Enable local block outputs** off and still benefit from memory reuse.

Dependency

Enabled by **Signal storage reuse**.

Command line parameter

`LocalBlockOutputs`

Recommended settings

Debugging	Clear
Traceability	No impact
Efficiency	Set
Safety precaution	No impact

More information

"Signals with Auto Storage Class"

Reuse block outputs

Specify signal storage memory usage

Checked (default)

- Real-Time Workshop reuses signal memory whenever possible, reducing stack size where signals are being buffered in local variables.
- Selecting this option trades code traceability for code efficiency.

Unchecked
Signals are stored in unique locations.

Dependency

Enabled by **Signal storage reuse**.

Command line parameter

`BufferReuse`

Recommended settings

Debugging	Clear
Traceability	Clear
Efficiency	Set
Safety precaution	No impact

More information

- "Signal Storage, Optimization, and Interfacing"
- "Signals with Auto Storage Class"

Ignore integer downcasts in folded expressions

Specify how Real-Time Workshop handles casting of intermediate variables in mixed-bit systems

Checked
Real-Time Workshop collapses block computations into a single expression, avoiding casts of intermediate variables, improving efficiency. Check this option if

- You are concerned with generating the least amount of code possible
- Code generation and simulation results do not need to match

Unchecked (default)
The results of 8- and 16-bit integer expressions are explicitly downcast.

Tip

Expressions involving 8- and 16-bit arithmetic are less likely to overflow in code than they are in simulation. Therefore, it is good practice to turn off **Ignore integer downcasts in folded expressions** for safety, to ensure that answers obtained from generated code are consistent with simulation results.

Command line parameter

`EnforceIntegerDowncast`

Recommended settings

Debugging	Clear
Traceability	No impact
Efficiency	Set
Safety precaution	Clear

More information

“Expression Folding Options”

Inline invariant signals

Transform symbolic names of invariant signals into constant values

Checked (default)
: Real-Time Workshop uses the numerical values of model parameters, instead of their symbolic names, in generated code.

Unchecked
: Uses symbolic names of model parameters in generated code.

For more information, see “Inline Invariant Signals”.

Tips

- An invariant signal is a block output signal that does not change.
- An *invariant signal* is not the same as an *invariant constant*. To inline invariant constants, select **Inline parameters**.

Dependency

Enabled by**Inline parameters**.

Command line parameter

`InlineInvariantSignals`

Recommended settings

Debugging	Clear
Traceability	Clear
Efficiency	Set
Safety precaution	No impact

More information

“Inline Invariant Signals”

Eliminate superfluous temporary variables (Expression folding)

Collapse block computations into single expressions

Checked (default)

- Enables expression folding.
- Eliminates temporary variables, incorporating the information into the main code statement.
- Improves code readability and efficiency.

Unchecked
Disables expression folding.

Dependency

Enabled by **Signal storage reuse**.

Command line parameter

`ExpressionFolding`

Recommended settings

Debugging	Clear
Traceability	No impact
Efficiency	Set
Safety precaution	No impact

More information

"Expression Folding"

Loop unrolling threshold

Specify minimum signal or parameter width for which a `for` loop is generated

Default: 5

Command line parameter

`RollThreshold`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	>0
Safety precaution	No impact

More information

"Loop Unrolling Threshold"

Remove code from floating-point to integer conversions that wraps out-of-range values

Remove wrapping code that handles out-of-range floating-point to integer conversion results

Checked

Removes code that ensures the execution of the generated code produces the same results as simulation when out-of-range conversions occur. Select this option if code efficiency is critical to your application and the following conditions are true for at least one block in the model:

- Computing the block's outputs or parameters involves converting floating-point data to integer or fixed-point data.
- The block's **Saturate on integer overflow** option is `disabled`.

Unchecked (default)

Out-of-range values simulation and generated code results match. The generated code is larger than when this option is checked.

Tips

- Enabling this option affects code generation results only for out-of-range values and hence cannot cause code generation results to differ from simulation results for in-range values.
- The code generator uses the `fmod` function to handle out-of-range conversion results.

Command line parameter

`EfficientFloat2IntCast`

Recommended settings

Debugging	Clear
Traceability	Clear
Efficiency	Set
Safety precaution	Clear

More information

"Remove Code from Floating-Point to Integer Conversions That Wraps Out-of-Range Values"

Diagnostics

Model Verification block enabling

Enable model verification blocks in the current model either globally or locally

Use local settings (default)
: Enables or disables blocks based on the value of the `Enable assertion` parameter of each block. If a block's `Enable assertion` parameter is on, the block is enabled; otherwise, the block is disabled.

Enable All
: Enables all model verification blocks in the model regardless of the settings of their `Enable assertion` parameters.

Disable All
: Disables all model verification blocks in the model regardless of the settings of their `Enable assertion` parameters.

Command line parameter

`AssertControl`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Diagnostics Options"

Hardware Implementation

Device type

Specify embedded hardware device

Selecting a device type specifies the hardware device to define your system's constraints:

- Default hardware properties appear as the initial values.
- Options with only one possible value cannot be changed.
- Options with more than one possible value provide a pulldown list of legal values.
- Static values are displayed in the table below. Options that you can modify are identified with an x.

Key:	word size = native word size							
	rounds to = Signed integer division rounds to							
	shift right = Shift right on a signed integer as arithmetic shift							
Device type	**Number of bits**					**Byte ordering**	**rounds to**	**shift right**
	char	**short**	**int**	**long**	**word size**			
Unspecified (assume 32–bit Generic) (default)	8	16	32	32	32	Un-specified	x	Set
Custom	x	x	x	x	x	x	x	x
32–bit Generic Embedded Processor	8	16	32	32	32	x	x	Set
32–bit PowerPC	8	16	32	32	32	Big Endian	Zero	Set
ARM 7/8/9	8	16	32	32	x	x	x	x
Freescale MPC5500	8	16	32	32	32	x	Zero	Set
Freescale 68332	8	16	32	32	32	Big Endian	x	Set
Infineon TriCore	8	16	32	32	32	Little Endian	x	Set
NEC V850	8	16	32	32	32	x	x	x
Renesas (Hitachi) SH-2, SH-4	8	16	32	32	32	x	x	x
TI C6000	8	16	32	40	32	x	Zero	Set
16–bit Generic Embedded Processor	8	16	16	32	16	x	x	Set
Renesas M16C	8	16	16	32	16	Little Endian	x	x
Freescale DSP563xx (16–bit mode)	8	16	16	32	16	x	x	Set

Key:	word size = native word size							
	rounds to = Signed integer division rounds to							
	shift right = Shift right on a signed integer as arithmetic shift							
Device type	**Number of bits**					**Byte ordering**	**rounds to**	**shift right**
	char	**short**	**int**	**long**	**word size**			
Freescale HC(S)12	8	16	16	32	16	Big Endian	x	Set
Infineon C16x, XC16x	8	16	16	32	16	Little Endian	Zero	Set
STMicroelectronics ST10	8	16	16	32	16	Little Endian	Zero	Set
TI C2000	16	16	16	32	16	x	Zero	Set
TI C5000	16	16	16	32	16	Big Endian	Zero	Set
8–bit Generic Embedded Processor	8	16	16	32	8	x	x	Set
8051 Compatible	8	16	16	32	8	x	x	Clear
Freescale 68HC11	8	16	16	32	8	Big Endian	x	Set
Freescale HC08	8	16	16	32	8	Big Endian	x	Set
32–bit Generic Real Time Simulator	8	16	32	32	32	x	x	Set
32–bit Real-Time Windows Target	8	16	32	32	32	Little Endian	Zero	Set
32–bit xPC Target (Intel Pentium)	8	16	32	32	32	Little Endian	x	Set
32–bit xPC Target (AMD Athlon)	8	16	32	32	32	Little Endian	x	Set

Key:	word size = native word size							
	rounds to = Signed integer division rounds to							
	shift right = Shift right on a signed integer as arithmetic shift							
Device type	**Number of bits**					**Byte ordering**	**rounds to**	**shift right**
	char	**short**	**int**	**long**	**word size**			
SGI UltraSPARC IIi	8	16	32	32	32	Big Endian	x	Set
ASIC/FPGA	NA	NA	NA	NA	NA	NA	NA	NA

Dependency

- Selecting **ASIC/FPGA** enables the **Emulation hardware (code generation only)** subpane.

For all other options, sets

- **char**
- **short**
- **int**
- **long**
- **native word size**
- **Byte ordering**
- **Signed integer division rounds to**
- **Shift right on a signed integer as arithmetic shift**

Command line parameter

`TargetHWDeviceType`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Device type"

Number of bits: char

Specify the character bit length

Default: 8

Minimum: 8

Maximum: 32

Tip

All values must be a multiple of 8.

Command line parameter

`TargetBitPerChar`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Target specific
Safety precaution	No impact

More information
"Number of bits"

Number of bits: short
Specify the data bit length

Default: 16

Minimum: 8

Maximum: 32

Tip
All values must be a multiple of 8.

Command line parameters
`TargetBitPerShort`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Target specific
Safety precaution	No impact

More information
"Number of bits"

Number of bits: int
Specify the data integer bit length

Default: 32

Minimum: 8

Maximum: 32

Tip
All values must be a multiple of 8.

Command line parameters
`TargetBitPerInt`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Target specific
Safety precaution	No impact

More information
"Number of bits"

Number of bits: long
Specify the data bit lengths

Default: 32

Minimum: 8

Maximum: 32

Tip
All values must be a multiple of 8.

Command line parameters

`TargetBitPerLong`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Target specific
Safety precaution	No impact

More information

"Number of bits"

Number of bits: native word size

Specify the microprocessor native word size

Default: 32

Minimum: 8

Maximum: 32

Tip

All values must be a multiple of 8.

Command line parameters

`TargetWordSize`

Recommended settings

Debugging	No impact
Traceability	No impact

Efficiency	Target specific
Safety precaution	No impact

More information

"Number of bits"

Byte ordering

Specify target hardware byte ordering

Big Endian
Specifies most significant byte first.

Little Endian
Specifies least significant byte first.

Unspecified (default)
Real-Time Workshop generates code that determines the endianness of the target; this is the least efficient option.

Command line parameter

`TargetEndianess`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Byte ordering"

Signed integer division rounds to

Specify how to produce a signed integer quotient

An ANSI C conforming compiler, used to compile code, rounds the result of dividing one signed integer by another based on the option selected:

Zero
: If the quotient is between two integers, the compiler chooses the integer that is closer to zero as the result.

Floor
: If the quotient is between two integers, the compiler chooses the integer that is closer to negative infinity.

Undefined (default)
: The compiler's rounding behavior is undefined if either or both operands are negative

Command line parameter

`TargetIntDivRoundTo`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Signed integer division rounds to"

Shift right on a signed integer as arithmetic shift

Specify how your compiler rounds the result of two signed integers

Checked (default)
Real-Time Workshop generates simple efficient code whenever the Simulink model performs arithmetic shifts on signed integers.

Unchecked
Real-Time Workshop generates fully portable but less efficient code to implement right arithmetic shifts.

Tips

- The preferred setting is to select this option if the C compiler implements a signed integer right shift as an arithmetic right shift.
- An arithmetic right shift fills bits vacated by the right shift with the value of the most significant bit, which indicates the sign of the number in twos complement notation.
-

Command line parameter

`TargetShiftRightIntArith`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	No impact

More information

“Shift right on a signed integer as arithmetic shift”

Emulation hardware (code generation only)

Specify current hardware characteristics

If the current hardware differs from the target hardware, you can generate code that runs on the current hardware but behaves as if it had been generated for and executed on the target hardware. The **Embedded hardware (simulation and code generation)** subpane specifies the target hardware properties. The **Emulation hardware (code generation only)** subpane is used to specify the current hardware properties.

Checked (default)
: The hardware used to test the code generated from the model is the same as the production hardware, or has the same characteristics.

Unchecked
: The hardware used to test the code generated from the model has different characteristics than the production hardware.

Dependency

Enables the `Emulation hardware` subpane.

Command line parameter

`ProdEqTarget`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

“Specifying Emulation Hardware Characteristics”

Real-Time Workshop (General)

System target file

Specify the system target file

Default: `grt.tlc`

You can specify the system target file in 2 ways:

- Use the System Target File Browser by clicking on 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. Click Apply or OK to configure for that target.

Tip

The System Target File Browser lists all system target files found on the MATLAB path. Using some of these might require additional licensed products, such as Real-Time Workshop Embedded Coder.

Command line parameter

`SystemTargetFile`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "System Target File"
- "Available Targets"

Language

Specify C or C++ code generation

C (default)
: Real-Time Workshop generates `.c` files and places the files in your build directory.

C++
: Real-Time Workshop 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 parameter

`TargetLang`

Recommended settings

Debugging	No impact
Traceability	No impact

Efficiency	No impact
Safety precaution	No impact

More information

- “Language”
- “Choosing and Configuring a Compiler”

Generate HTML report

Document generated code in an HTML report

Checked
Generates a navigable summary of code generation source files in an HTML report and places the 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).

Unchecked (default)
Summary of files not generated.

Dependency

Enables **Launch report automatically**.

Command line parameter

`GenerateReport`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

- "Generate HTML Report"
- "Viewing Generated Code in Generated HTML Reports"

Launch report automatically

Specify automatically displaying HTML reports

Checked (default when enabled)
: The HTML summary and index are automatically loaded into a new browser window and displayed.

Unchecked
: The HTML report is not opened, but is still available in the `html` directory.

Dependency

Enabled by **Generate HTML Report**.

Command line parameter

`LaunchReport`

Recommended settings

Debugging	Set
Traceability	Set

Efficiency	No impact
Safety precaution	No impact

More information

"Generate HTML Report"

TLC options

Specify additional TLC options

You can enter TLC command line options and arguments.

Tips

- Specifying TLC options does not add flags to the **Make Command** field.
- The Generated HTML Report summary section lists TLC options specified for the build in which the report is generated.

Command line parameter

`TLCOptions`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "Command-Line Arguments"
- "TLC Options"

Generate makefile

Specify generation of a makefile

Checked (default)
Generates a makefile for a model during the build process.

Unchecked
Suppress the generation of a makefile. When you clear this option you must set up any post code generation build processing, including compilation and linking, as a user-defined command.

Dependencies

Clearing this option disables the following options:

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

Command line parameter

`GenerateMakefile`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Generate Makefile”
- “Customizing Post Code Generation Build Processing”

Make command

Specify `make` command

Default: `make_rtw`

The make command, a high-level M-file command, invoked when a build is initiated, 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.
- Third-party targets might supply a `make` command. See the vendor's documentation.
- Arguments can be specified in this field, and are passed into the makefile-based build process.

Tip

Most targets use the default command.

Dependency

Enabled by **Generate makefile**.

Command line parameter

`MakeCommand`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "Make Command"
- "Template Makefiles and Make Options"

Template makefile

Specify template makefile

Default: grt_default_tmf

The template makefile determines which compiler runs, during the make phase of the build, to compile the generated code. There are two ways to specify template makefiles:

- 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 a filename extension is not included 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 providing your own template makefile.

Dependency

Enabled by **Generate makefile**.

Command line parameter

`TemplateMakefile`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "Template Makefile"
- "Template Makefiles and Make Options"
- "Available Targets"

Generate code only

Specify code generation and executable build

Checked
The build process generates code and a `make` file, but does not invoke the `make` command. When you select this option, the caption of the **Build** button changes to **Generate code**.

Unchecked (default)
The build process generates and compiles code, and an executable is built.

Dependencies

- Changes **Build/Generate code** button based on setting.
- Generates a `make` file only if **Generate makefile** option is checked.

Command line parameter

`GenCodeOnly`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Generate Code Only”
- “Controlling the Compiling and Linking Phases of the Build Process”

Build/Generate code

Initiate build process

Provides one way of initiating the build process for a model or subsystem. When you check the **Generate code only** option, the caption of the **Build** button changes to **Generate code**.

Dependency

Operation based on **Generate code only** setting.

Command line parameter

`GenCodeOnly`

Recommended settings

Debugging	Build
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

“Build Button”

Comments

Include comments

Specify what comments are in generated files

Checked (default)
: Comments are placed in the generated files based on the selections in the **Auto generated comments** pane.

Unchecked
: Comments do not appear in the generated files.

Dependencies

Enables the following **Auto generated comments pane** options:

- **Simulink block comments**
- **Show eliminated blocks**
- **Verbose comments for SimulinkGlobal storage class**

Command line parameter

`GenerateComments`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

"Include Comments"

Simulink block comments

Insert Simulink block comments

Checked (default)
: Automatically generated comments that describe a block's code precede that code in the generated file.

Unchecked
: No comments are inserted.

Dependency

Enabled by **Include comments**.

Command line parameter

`SimulinkBlockComments`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

"Simulink Block Comments"

Show eliminated blocks

Insert eliminated blocks comments

Checked
Statements pertaining to blocks that were eliminated as the result of optimizations (such as parameter inlining) appear as comments in the generated code.

Unchecked (default)
No statements are inserted.

Dependency
Enabled by **Include comments**.

Command line parameter
`ShowEliminatedStatements`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information
"Show Eliminated Blocks"

Verbose comments for SimulinkGlobal storage class
Generate comments in model parameter structure declaration

Controls 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.

Checked
Parameter comments are generated regardless of the number of parameters.

Unchecked (default)
: Parameter comments are generated if less than 1000 parameters are declared. This reduces the size of the generated file for models with a large number of parameters.

Dependency

Enabled by **Include comments**.

Command line parameter

`ForceParamTrailComments`

Recommended settings

Debugging	Set
Traceability	Set
Efficiency	No impact
Safety precaution	No impact

More information

"Verbose Comments for SimulinkGlobal Storage Class"

Symbols

Maximum identifier length

Specify maximum number of characters in generated function, type definition, variable names

Default: 31

Minimum: 31

Maximum: 256

Allows you to limit the number of characters in function, type definition, and variable names.

- 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.
- Must be the same for both top and referenced models.

Command line parameter

`MaxIdLength`

Recommended settings

Debugging	Set
Traceability	>30
Efficiency	No impact
Safety precaution	No impact

More information

- “Maximum Identifier Length”
- “Parameterizing Referenced Models”

Custom Code

Source file

Specify code appearing at top of generated file

Code is placed near the top of the generated *model*.c or *model*.cpp file, outside of any function.

Command line parameter

`CustomSource`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Custom Code Options"

Header file

Specify code appearing near top of generated file

Code is placed near the top of the generated *model*.h header file.

Command line parameter

`CustomHeaderCode`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Custom Code Options"

Initialize function

Specify code appearing in initialize function

Code is placed inside the model's initialize function in the *model*.c or *model*.cpp file.

Command line parameter

`CustomInitializer`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Custom Code Options"

Terminate function

Specify code appearing in terminate function

Specify code to appear in the model's generated terminate function in the *model*.c or *model*.cpp file.

Dependency

You should also select the **Terminate function required** check box on the **Real-Time Workshop > Interface** pane.

Command line parameter

`CustomTerminator`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Custom Code Options"

Include directories

Specify list of include directories

Specify a space-separated list of include directories to be added 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 parameter

`CustomInclude`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Custom Code Options"

Source files

Specify list of source files

- Specify a space-separated list of source files to be compiled and linked with the generated code.
- The filename is sufficient if the file is in the current MATLAB directory or in one of the Include directories.

Command line parameter

`CustomSourceCode`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Custom Code Options"

Libraries

Specify list of additional libraries

List of additional libraries to be linked with. The libraries can be specified with a full path or just a filename when located in the current MATLAB directory or is listed as one of the Include directories.

Command line parameter

`CustomLibrary`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Custom Code Options"

Debug

Verbose build

Display code generation progress

Checked (default)
: The MATLAB Command Window displays progress information indicating code generation stages and compiler output during code generation.

Unchecked
: No progress information is displayed.

Command line parameter

`RTWVerbose`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	No impact
Safety precaution	Set

More information

"Verbose Build"

Retain .rtw file

Specify *model*.rtw file retention

Checked
The *model*.rtw file is retained in the current build directory. This option is useful if you are modifying the target files and need to look at the file.

Unchecked (default)
The *model*.rtw is deleted from the build directory at the end of the build process.

Command line parameter

`RetainRTWFile`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Retain .rtw File"

Profile TLC

Profile execution time of TLC files

Checked
The TLC profiler analyzes the performance of TLC code executed during code generation, and generates an HTML report.

Unchecked (default)
The performance is not profiled.

Command line parameter

`ProfileTLC`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Profile TLC"

Start TLC debugger when generating code

Specify use of TLC debugger

Checked
The TLC debugger starts during code generation.

Unchecked (default)
The TLC debugger is not started.

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 parameter

`TLCDebug`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Start TLC Debugger When Generating Code"

Start TLC coverage when generating code

Generate TLC execution report

Checked
Generates `.log` files containing the number of times each line of TLC code is executed during code generation.

Unchecked (default)
No report is generated.

Tip

You can also generate the TLC execution report by entering the `-dg` argument into the **System target file** field.

Command line parameter

`TLCCoverage`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Start TLC Coverage When Generating Code"

Enable TLC assertion

Produce TLC stack trace

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

Unchecked (default)
TLC assertion code is ignored.

Command line parameter

`TLCAssert`

Recommended settings

Debugging	Set
Traceability	No impact
Efficiency	No impact
Safety precaution	Set

More information

"Enable TLC Assertion"

Interface

Target floating-point math environment

Specify floating-point math library extension

C89/C90 (ANSI) (default)
: 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`.

Tips

- Before setting this option, verify that your compiler supports the library you want to use. If you select an option value that your compiler does not support, compiler errors can occur.
- **Restriction** — Stateflow supports only C89/C90(ANSI). Selecting a different option has no effect on code generated for Stateflow components.

Command line parameter

`GenFloatMathFcnCalls`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	Set
Safety precaution	No impact

More information

"Target Floating-Point Math Environment"

Utility function generation

Specify utility functions generation location

Auto (default)
: 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 parameter

`UtilityFuncGeneration`

Recommended settings

Debugging	Shared
Traceability	Shared
Efficiency	Shared
Safety precaution	No impact

More information

"Utility Function Generation"

MAT-file variable name modifier

Select string added to MAT-file variable names

rt_ (default)
: Adds a prefix string.

_rt
: Adds a suffix string.

none
: Does not add a string.

Command line parameter

`LogVarNameModifier`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"MAT-File Variable Name Modifier"

Interface

Specify included data interface (API)

None (default)
: API is not included in generated code.

C-API
: Use `C-API` data interface.

External mode
: Use external data interface.

ASAP2
: Use `ASAP2` data interface.

Dependencies

The following are enabled by selecting **C-API**

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

The following are enabled by selecting **External mode**

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

Command line parameter

`GenerateASAP2`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "Interface"
- "Interface Option Dependencies"

Signals in C API

Generate C API signal structure

Checked (default)
Generates C API for global block outputs.

Unchecked
Does not generate C API signals.

Dependency

Enabled by selecting **C-API** data exchange interface.

Command line parameter

`RTWCAPISignals`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Generating C-API Files"

Parameters in C API

Generate C API parameter tuning structures

Checked (default)
: Generate C API for global block and model parameters.

Unchecked
: Do not generate C API parameters.

Dependency

Enabled by selecting **C-API** data exchange interface.

Command line parameter

`RTWCAPIParams`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

"Generating C-API Files"

Transport layer

Specify transport protocols for external mode communications

tcpip (default)
: Use a TCP/IP transport mechanism.

serial_win32
Use a serial transport mechanism.

Tip
The external interface MEX-file being used is not editable, it is specified in `extmode_transports.m`.

Dependency
Enabled by selecting **External mode** data exchange interface.

Command line parameter
`ExtModeTransport`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- “Creating an External Mode Communication Channel”
- “Target Interfacing”

MEX-file arguments
Specify external mode MEX arguments.

For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- The network name of your target
- A TCP/IP server port number
- Verbosity level (0 or 1)

For a serial transport, `ext_serial_win32_comm` allows 3 optional arguments:

- Verbosity level (0 or 1)
- Serial port ID
- Baud rate (selected from the set 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, with a default of 57600)

Dependency

Enabled by selecting **External mode** data exchange interface.

Command line parameter

`ExtModeMexArgs`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

- "Target Interfacing"
- "Client/Server Implementations"

Static memory allocation

Control memory buffer for external mode communication

Unchecked (default)
: Use a static memory buffer for external mode instead of allocating dynamic memory (calls to malloc).

Checked

Enables **Static memory buffer size** parameter. Enter number of bytes to preallocate for external mode communications buffers in the target. The default value is 1,000,000 bytes.

Tips

- If you enter too small a value for your application, external mode issues and out-of-memory error.
- To determine how much memory you need to allocate, enable verbose mode on the target to display the amount of memory it tries to allocate and is available.

Dependencies

- Enabled by selecting **External mode** data exchange interface.
- Enables **Static memory buffer size**.

Command line parameter

`ExtModeStaticAlloc`

Recommended settings

Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

More information

“External Mode Interface Options”

Index

A

B

C

D

E

F

G

T

U