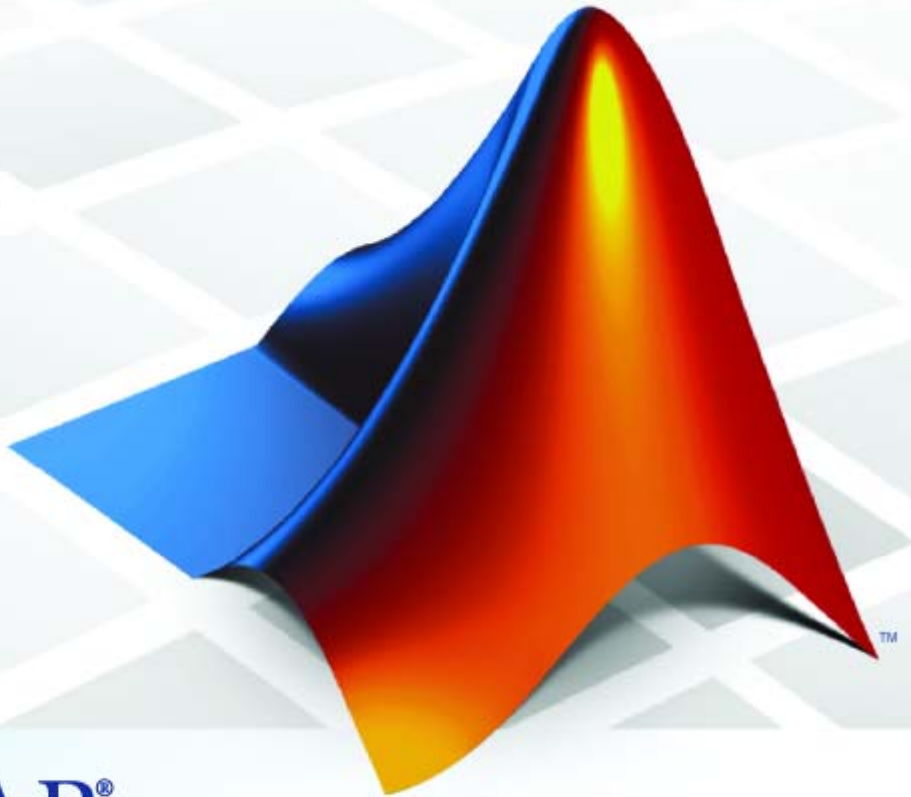


Real-Time Workshop[®] 7

Reference



MATLAB[®]
& **SIMULINK[®]**

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

© COPYRIGHT 2006–2009 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

| | | |
|----------------|-------------|---|
| March 2006 | Online only | New for Version 6.4 |
| September 2006 | Online only | Revised for Version 6.5 (Release 2006b) |
| March 2007 | Online only | Revised for Version 6.6 (Release 2007a) |
| September 2007 | Online only | Revised for Version 7.0 (Release 2007b) |
| March 2008 | Online only | Revised for Version 7.1 (Release 2008a) |
| October 2008 | Online only | Revised for Version 7.2 (Release 2008b) |
| March 2009 | Online only | Revised for Version 7.3 (Release 2009a) |
| September 2009 | Online only | Revised for Version 7.4 (Release 2009b) |

Product Limitations Summary

1

Glossary

Function Reference

2

| | |
|--|-----|
| Build Information | 2-2 |
| Build Process | 2-4 |
| Embedded MATLAB Code Generation | 2-5 |
| Project Documentation | 2-6 |
| Rapid Simulation | 2-6 |
| Target Language Compiler and Function Library | 2-6 |

Alphabetical List

3

Block Reference

4

| | |
|----------------------------------|-----|
| Custom Code | 4-2 |
| Interrupt Templates | 4-3 |
| S-Function Target | 4-4 |
| VxWorks | 4-5 |

Blocks — Alphabetical List

5

Configuration Parameters for Simulink Models

6

| | |
|--|------|
| Real-Time Workshop Pane: General | 6-2 |
| Real-Time Workshop: General Tab Overview | 6-5 |
| System target file | 6-6 |
| Language | 6-8 |
| Compiler optimization level | 6-10 |
| Custom compiler optimization flags | 6-12 |
| TLC options | 6-13 |
| Generate makefile | 6-15 |
| Make command | 6-17 |
| Template makefile | 6-19 |
| Ignore custom storage classes | 6-21 |
| Ignore test point signals | 6-23 |

| | |
|---|-------------|
| Select objective | 6-25 |
| Prioritized objectives | 6-27 |
| Set objectives | 6-28 |
| Configuration Set Objectives Dialog Box | 6-29 |
| Check model | 6-31 |
| Check model before generating code | 6-32 |
| Generate code only | 6-34 |
| Build/Generate code | 6-36 |
| | |
| Real-Time Workshop Pane: Report | 6-37 |
| Real-Time Workshop: Report Tab Overview | 6-39 |
| Create code generation report | 6-40 |
| Launch report automatically | 6-43 |
| Code-to-model | 6-45 |
| Model-to-code | 6-47 |
| Configure | 6-49 |
| Eliminated / virtual blocks | 6-50 |
| Traceable Simulink blocks | 6-52 |
| Traceable Stateflow objects | 6-54 |
| Traceable Embedded MATLAB functions | 6-56 |
| | |
| Real-Time Workshop Pane: Comments | 6-58 |
| Real-Time Workshop: Comments Tab Overview | 6-60 |
| Include comments | 6-61 |
| Simulink block / Stateflow object comments | 6-62 |
| Show eliminated blocks | 6-63 |
| Verbose comments for SimulinkGlobal storage class | 6-64 |
| Simulink block descriptions | 6-65 |
| Simulink data object descriptions | 6-67 |
| Custom comments (MPT objects only) | 6-69 |
| Custom comments function | 6-71 |
| Stateflow object descriptions | 6-73 |
| Requirements in block comments | 6-75 |
| | |
| Real-Time Workshop Pane: Symbols | 6-77 |
| Real-Time Workshop: Symbols Tab Overview | 6-81 |
| Global variables | 6-82 |
| Global types | 6-84 |
| Field name of global types | 6-87 |
| Subsystem methods | 6-89 |
| Subsystem method arguments | 6-92 |
| Local temporary variables | 6-94 |
| Local block output variables | 6-96 |

| | |
|--|--------------|
| Constant macros | 6-98 |
| Minimum mangle length | 6-100 |
| Maximum identifier length | 6-102 |
| Generate scalar inlined parameter as | 6-104 |
| Signal naming | 6-105 |
| M-function | 6-107 |
| Parameter naming | 6-109 |
| #define naming | 6-111 |
| Use the same reserved names as Simulation Target | 6-113 |
| Reserved names | 6-114 |
| Real-Time Workshop Pane: Custom Code | 6-116 |
| Real-Time Workshop: Custom Code Tab Overview | 6-118 |
| Use the same custom code settings as Simulation Target .. | 6-119 |
| Use local custom code settings (do not inherit from main model) | 6-120 |
| Source file | 6-122 |
| Header file | 6-123 |
| Initialize function | 6-124 |
| Terminate function | 6-125 |
| Include directories | 6-126 |
| Source files | 6-128 |
| Libraries | 6-130 |
| Real-Time Workshop Pane: Debug | 6-132 |
| Real-Time Workshop: Debug Tab Overview | 6-134 |
| Verbose build | 6-135 |
| Retain .rtw file | 6-136 |
| Profile TLC | 6-137 |
| Start TLC debugger when generating code | 6-138 |
| Start TLC coverage when generating code | 6-139 |
| Enable TLC assertion | 6-140 |
| Real-Time Workshop Pane: Interface | 6-141 |
| Real-Time Workshop: Interface Tab Overview | 6-145 |
| Target function library | 6-146 |
| Utility function generation | 6-148 |
| Support: floating-point numbers | 6-150 |
| Support: absolute time | 6-152 |
| Support: non-finite numbers | 6-154 |
| Support: continuous time | 6-156 |
| Support: complex numbers | 6-158 |
| Support: non-inlined S-functions | 6-160 |

| | |
|---|--------------|
| Support: variable-size signals | 6-162 |
| Multiword type definitions | 6-163 |
| Maximum word length | 6-165 |
| GRT compatible call interface | 6-167 |
| Single output/update function | 6-169 |
| Terminate function required | 6-171 |
| Generate reusable code | 6-173 |
| Reusable code error diagnostic | 6-176 |
| Pass root-level I/O as | 6-178 |
| Block parameter visibility | 6-180 |
| Internal data visibility | 6-182 |
| Block parameter access | 6-184 |
| Internal data access | 6-186 |
| External I/O access | 6-188 |
| Generate destructor | 6-190 |
| Use operator new for referenced model object registration | 6-192 |
| Generate preprocessor conditionals | 6-194 |
| Suppress error status in real-time model data structure .. | 6-196 |
| Configure Model Functions | 6-198 |
| Configure C++ Encapsulation Interface | 6-199 |
| Create Simulink (S-Function) block | 6-200 |
| Enable portable word sizes | 6-202 |
| MAT-file logging | 6-204 |
| MAT-file variable name modifier | 6-206 |
| Interface | 6-208 |
| Signals in C API | 6-211 |
| Parameters in C API | 6-212 |
| Transport layer | 6-213 |
| MEX-file arguments | 6-215 |
| Static memory allocation | 6-217 |
| Static memory buffer size | 6-219 |
| | |
| Real-Time Workshop Pane: RSim Target | 6-221 |
| Real-Time Workshop: RSim Target Tab Overview | 6-223 |
| Enable RSim executable to load parameters from a MAT-file | 6-224 |
| Solver selection | 6-225 |
| Force storage classes to AUTO | 6-226 |
| | |
| Real-Time Workshop Pane: Real-Time Workshop S-Function Code Generation Options | 6-227 |

| | |
|---|--------------|
| Real-Time Workshop S-Function Code Generation Options | |
| Tab Overview | 6-229 |
| Create new model | 6-230 |
| Use value for tunable parameters | 6-231 |
| Include custom source code | 6-232 |
| Real-Time Workshop Pane: Tornado Target | 6-233 |
| Real-Time Workshop: Tornado Target Tab Overview | 6-235 |
| Target function library | 6-236 |
| Utility function generation | 6-238 |
| MAT-file logging | 6-239 |
| MAT-file variable name modifier | 6-241 |
| Code Format | 6-243 |
| StethoScope | 6-244 |
| Download to VxWorks target | 6-246 |
| Base task priority | 6-248 |
| Task stack size | 6-250 |
| External mode | 6-251 |
| Transport layer | 6-253 |
| MEX-file arguments | 6-255 |
| Static memory allocation | 6-257 |
| Static memory buffer size | 6-259 |
| Parameter Reference | 6-261 |
| Recommended Settings Summary | 6-261 |
| Parameter Command-Line Information Summary | 6-285 |

Configuration Parameters for Embedded MATLAB Coder

7

| | |
|---|------------|
| Real-Time Workshop Dialog Box for Embedded | |
| MATLAB Coder | 7-2 |
| Real-Time Workshop Dialog Box Overview | 7-2 |
| General Tab | 7-3 |
| Report Tab | 7-6 |
| Symbols Tab | 7-7 |
| Custom Code Tab | 7-13 |
| Debug Tab | 7-16 |
| Interface Tab | 7-18 |

| | |
|--------------------------|------|
| Advanced Tab | 7-23 |
| Generate code only | 7-24 |

Automatic C MEX Generation Dialog Box for Embedded

| | |
|--|------|
| MATLAB Coder | 7-25 |
| Automatic C MEX Generation Dialog Box Overview | 7-25 |
| General Tab | 7-26 |
| Report Tab | 7-30 |
| Symbols Tab | 7-32 |
| Custom Code Tab | 7-34 |
| Advanced Tab | 7-37 |

Hardware Implementation Dialog Box for Embedded

| | |
|---|------|
| MATLAB Coder | 7-40 |
| Hardware Implementation Parameters Dialog Box Overview | 7-40 |
| Hardware Implementation Parameters | 7-42 |

Compiler Options Dialog Box

| | |
|---|------|
| Compiler Options Parameters Dialog Box Overview | 7-49 |
| Compiler Options Parameters | 7-51 |

Model Advisor Checks

8

| | |
|---|------|
| Real-Time Workshop Checks | 8-2 |
| Real-Time Workshop Overview | 8-3 |
| Check solver for code generation | 8-4 |
| Identify questionable blocks within the specified system .. | 8-6 |
| Check for model reference configuration mismatch | 8-7 |
| Check the hardware implementation | 8-8 |
| Identify questionable software environment specifications | 8-10 |
| Identify questionable code instrumentation (data I/O) | 8-12 |
| Check for blocks that have constraints on tunable parameters | 8-13 |
| Identify questionable subsystem settings | 8-15 |
| Disable signal logging | 8-16 |

| | |
|--|------|
| Identify blocks that generate expensive saturation and rounding code | 8-17 |
| Check sample times and tasking mode | 8-18 |
| Identify questionable fixed-point operations | 8-19 |
| Check model configuration settings against code generation objectives | 8-27 |

Index

Product Limitations Summary

The following topics identify Real-Time Workshop® feature limitations:

- “C++ Target Language Limitations”
- “packNGo Function Limitations”
- “Tunable Expression Limitations”
- “Limitations on Specifying Data Types in the Workspace Explicitly”
- “Code Reuse Limitations”
- “Real-Time Workshop Model Referencing Limitations”
- “External Mode Limitations”
- “Noninlined S-Function Parameter Type Limitations”
- “S-Function Target Limitations”
- “Rapid Simulation Target Limitations”
- “Asynchronous Support Limitations”
- “C API Limitations”
- “Block Support Considerations”

application modules

With respect to Real-Time Workshop program architecture, these are collections of programs that implement functions carried out by the system-dependent, system-independent, and application components.

atomic subsystem

Subsystem whose blocks are executed as a unit before moving on. Conditionally executed subsystems are atomic, and atomic subsystems are nonvirtual. Unconditionally executed subsystems are virtual by default, but can be designated as atomic. The Real-Time Workshop build process can generate reusable code only for nonvirtual subsystems.

base sample rate

Fundamental sample time of a model; in practice, limited by the fastest rate at which a processor's timer can generate interrupts. All sample times must be integer multiples of the base rate.

block I/O structure (model_B)

Global data structure for storing block output signals. The number of block output signals is the sum of the widths of the data output ports of all nonvirtual blocks in your model. By default, Simulink® and the Real-Time Workshop build process try to reduce the size of the *model_B* structure by reusing the entries in the *model_B* structure and making other entries local variables.

block target file

File that describes how a specific Simulink block is to be transformed to a language such as C, based on the block's description in the Real-Time Workshop file (*model.rtw*). Typically, there is one block target file for each Simulink block.

code reuse

Optimization whereby code generated for identical nonvirtual subsystems is collapsed into one function that is called for each subsystem instance with appropriate parameters. Code reuse, along with *expression folding*, can dramatically reduce the amount of generated code.

configuration

Set of attributes for a model which defines parameters governing how a model simulates and generates code. A model can have one or more such configuration sets, and users can switch between them to change code generation targets or to modify the behavior of models in other ways.

configuration component

Named element of a configuration set. Configuration components encapsulate settings associated with the **Solver**, **Data Import/Export**, **Optimization**, **Diagnostics**, **Hardware Implementation**, **Model Referencing**, and **Real-Time Workshop** panes in the Configuration Parameters dialog box. A component may contain subcomponents.

embedded real-time (ERT) target

Target configuration that generates model code for execution on an independent embedded real-time system. Requires a Real-Time Workshop® Embedded Coder™ license.

expression folding

Code optimization technique that minimizes the computation of intermediate results at block outputs and the storage of such results in temporary buffers or variables. It can dramatically improve the efficiency of generated code, achieving results that compare favorably with hand-optimized code.

file extensions

The table below lists the Simulink, Target Language Compiler, and Real-Time Workshop file extensions.

| Extension | Created by | Description |
|------------------|--------------------------|--|
| .c or .cpp | Target Language Compiler | The generated C or C++ code |
| .h | Target Language Compiler | C/C++ include header file used by the .c or .cpp program |

| Extension | Created by | Description |
|-----------|--|--|
| .mdl | Simulink | Contains structures associated with Simulink block diagrams |
| .mk | Real-Time Workshop | Makefile specific to your model that is derived from the template makefile |
| .rtw | Real-Time Workshop | Intermediate compilation (<i>model.rtw</i>) of a .mdl file used in generating C or C++ code |
| .tlc | The MathWorks and Real-Time Workshop users | Target Language Compiler script files that the Real-Time Workshop build process uses to generate code for targets and blocks |
| .tmf | Supplied with Real-Time Workshop | Template makefiles |
| .tmw | Real-Time Workshop | Project marker file inside a build directory that identifies the date and product version of generated code |

generic real-time (GRT) target

Target configuration that generates model code for a real-time system, with the resulting code executed on your workstation. (Execution is not tied to a real-time clock.) You can use GRT as a starting point for targeting custom hardware.

host system

Computer system on which you create and may compile your real-time application. Also referred to as emulation hardware.

inline

Generally, this means to place something directly in the generated source code. You can inline parameters and S-functions using the Real-Time Workshop software and the Target Language Compiler.

inlined parameters

(Target Language Compiler Boolean global variable: `InlineParameters`)
The numerical values of the block parameters are hard-coded into the generated code. Advantages include faster execution and less memory use, but you lose the ability to change the block parameter values at run time.

inlined S-function

An S-function can be inlined into the generated code by implementing it as a `.t1c` file. The code for this S-function is placed in the generated model code itself. In contrast, noninlined S-functions require a function call to an S-function residing in an external MEX-file.

interrupt service routine (ISR)

Piece of code that your processor executes when an external event, such as a timer, occurs.

loop rolling

(Target Language Compiler global variable: `RollThreshold`) Depending on the block's operation and the width of the input/output ports, the generated code uses a `for` statement (rolled code) instead of repeating identical lines of code (flat code) over the signal width.

make

Utility to maintain, update, and regenerate related programs and files. The commands to be executed are placed in a *makefile*.

makefiles

Files that contain a collection of commands that allow groups of programs, object files, libraries, and so on, to interact. Makefiles are executed by your development system's make utility.

model.rtw

Intermediate record file into which the Real-Time Workshop build process compiles the blocks, signals, states, and parameters a model, which the Target Language Compiler reads to generate code to represent the model.

multitasking

Process by which a microprocessor schedules the handling of multiple tasks. In generated code, the number of tasks is equal to the number of sample times in your model. *See also* pseudo multitasking.

noninlined S-function

In the context of the Real-Time Workshop build process, this is any C MEX S-function that is not implemented using a customized `.tlc` file. If you create a C MEX S-function as part of a Simulink model, it is by default noninlined unless you write your own `.tlc` file that inlines it.

nonreal-time

Simulation environment of a block diagram provided for high-speed simulation of your model. Execution is not tied to a real-time clock.

nonvirtual block

Any block that performs some algorithm, such as a Gain block. The Real-Time Workshop build process generates code for all nonvirtual blocks, either inline or as separate functions and files, as directed by users.

pseudo multitasking

On processors that do not offer *multitasking* support, you can perform pseudo multitasking by scheduling events on a fixed time sharing basis.

real-time model data structure

The Real-Time Workshop build process encapsulates information about the root model in the real-time model data structure, often abbreviated as `rtM`. `rtM` contains global information related to timing, solvers, and logging, and model data such as inputs, outputs, states, and parameters.

real-time system

Computer that processes real-world events as they happen, under the constraint of a real-time clock, and that can implement algorithms in

dedicated hardware. Examples include mobile telephones, test and measurement devices, and avionic and automotive control systems.

Real-Time Workshop target

Set of code files generated by the Real-Time Workshop build process for a standard or custom target, specified by a Real-Time Workshop configuration component. These source files can be built into an executable program that will run independently of Simulink. *See also* simulation target, configuration.

run-time interface

Wrapper around the generated code that can be built into a stand-alone executable. The run-time interface consists of routines to move the time forward, save logged variables at the appropriate time steps, and so on. The run-time interface is responsible for managing the execution of the real-time program created from your Simulink block diagram.

S-function

Customized Simulink block written in C, Fortran, or M-code. The Real-Time Workshop build process can target C code S-functions as is or users can *inline* C code S-functions by preparing TLC scripts for them.

simstruct

Simulink data structure and associated application program interface (API) that enables S-functions to communicate with other entities in models. Simstructs are included in code generated by the Real-Time Workshop build process for noninlined S-functions.

simulation target

Set of code files generated for a model which is referenced by a Model block. Simulation target code is generated into `/slprj/sim` project directory in the working directory. Also an executable library compiled from these codes that implements a Model block. *See also* Real-Time Workshop target.

single-tasking

Mode in which a model runs in one task, regardless of the number of sample rates it contains.

stiffness

Property of a problem that forces a numerical method, in one or more intervals of integration, to use a step length that is excessively small in relation to the smoothness of the exact solution in that interval.

system target file

Entry point to the Target Language Compiler program, used to transform the Real-Time Workshop file into target-specific code.

target file

File that is compiled and executed by the Target Language Compiler. The block and system target TLC files used specify how to transform the Real-Time Workshop file (*model.rtw*) into target-specific code.

Target Language Compiler (TLC)

Program that compiles and executes system and target files by translating a *model.rtw* file into a target language by means of TLC scripts and template makefiles.

Target Language Compiler program

One or more TLC script files that describe how to convert a *model.rtw* file into generated code. There is one TLC file for the target, plus one for each built-in block. Users can provide their own TLC files to inline S-functions or to wrap existing user code.

target system

Specific or generic computer system on which your real-time application is intended to execute. Also referred to as embedded hardware.

targeting

Process of creating software modules appropriate for execution on your target system.

task identifier (tid)

In generated code, each sample rate in a multirate model is assigned a task identifier (tid). The tid is used by the model output and update routines to control the portion of your model that should execute at a given time step. Single-rate systems ignore the tid. *See also* base sample rate.

template makefile

Line-for-line makefile used by a make utility. The Real-Time Workshop build process converts the template makefile to a makefile by copying the contents of the template makefile (usually `system.tmf`) to a makefile (usually `system.mk`) replacing tokens describing your model's configuration.

virtual block

Connection or graphical block, for example a Mux block, that has no algorithmic functionality. Virtual blocks incur no real-time overhead as no code is generated for them.

work vector

Data structures for saving internal states or similar information, accessible to blocks that may require such work areas. These include state work (`rtDWork`), real work (`rtRWork`), integer work (`rtIWork`), and pointer work (`rtPWork`) structures. For example, the Memory block uses a real work element for each signal.

Function Reference

Build Information (p. 2-2)

Set up and manage model's build information

Build Process (p. 2-4)

Perform build process steps

Embedded MATLAB Code Generation (p. 2-5)

Generate embeddable C code or MEX code from M-file

Project Documentation (p. 2-6)

Document generated code

Rapid Simulation (p. 2-6)

Get model's parameter structures

Target Language Compiler and Function Library (p. 2-6)

Optimize code generated for model's blocks

Build Information

| | |
|-------------------------------|--|
| <code>addCompileFlags</code> | Add compiler options to model's build information |
| <code>addDefines</code> | Add preprocessor macro definitions to model's build information |
| <code>addIncludeFiles</code> | Add include files to model's build information |
| <code>addIncludePaths</code> | Add include paths to model's build information |
| <code>addLinkFlags</code> | Add link options to model's build information |
| <code>addLinkObjects</code> | Add link objects to model's build information |
| <code>addNonBuildFiles</code> | Add nonbuild-related files to model's build information |
| <code>addSourceFiles</code> | Add source files to model's build information |
| <code>addSourcePaths</code> | Add source paths to model's build information |
| <code>addTMFTokens</code> | Add template makefile (TMF) tokens that provide build-time information for makefile generation |
| <code>findIncludeFiles</code> | Find and add include (header) files to build information object |
| <code>getCompileFlags</code> | Compiler options from model's build information |
| <code>getDefines</code> | Preprocessor macro definitions from model's build information |
| <code>getFullFileList</code> | All files from model's build information |
| <code>getIncludeFiles</code> | Include files from model's build information |

| | |
|---|--|
| <code>getIncludePaths</code> | Include paths from model's build information |
| <code>getLinkFlags</code> | Link options from model's build information |
| <code>getNonBuildFiles</code> | Nonbuild-related files from model's build information |
| <code>getSourceFiles</code> | Source files from model's build information |
| <code>getSourcePaths</code> | Source paths from model's build information |
| <code>packNGo</code> | Package model code in zip file for relocation |
| <code>RTW.getBuildDir</code> | Build directory information for specified model |
| <code>switchTarget</code> | Specify target for configuration set |
| <code>updateFilePathsAndExtensions</code> | Update files in model's build information with missing paths and file extensions |
| <code>updateFileSeparator</code> | Change file separator used in model's build information |

Build Process

rtwbuild

Initiate build process

rtwrebuild

Rebuild generated code

rtw_precompile_libs

Build libraries within model without building model

Embedded MATLAB Code Generation

`emlc`

Generate embeddable C code from
MATLAB® code

Project Documentation

| | |
|-----------|--|
| rtwreport | Generate report documenting the generated code for a model |
| rtwtrace | Trace block to generated code |

Rapid Simulation

| | |
|-----------------|---|
| rsimgetrtp | Global model parameter structure |
| rsimsetrtpparam | Set parameters of rtP model parameter structure |

Target Language Compiler and Function Library

| | |
|-----|---|
| tlc | Invokes Target Language Compiler to convert model description file to generated code. |
|-----|---|

See the “TLC Function Library Reference” in the Real-Time Workshop Target Language Compiler documentation for a list of Target Language Compiler 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 <i>options</i> . Available for nonmakefile build environments only. |

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

- Target-independent values `Optimizations on` (faster runs) and `Optimizations off` (faster builds), which allow you to easily toggle compiler optimizations on and off during code development
- The value `Custom` for entering custom compiler optimization flags at Simulink GUI level (rather than at other levels of the build process)

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

Description

The `addCompileFlags` function adds specified compiler options to the model's build information. Real-Time Workshop stores the compiler

addCompileFlags

options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

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

Examples

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

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

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

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


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

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

See Also

`addDefines`, `addLinkFlags`
“Programming a Post Code Generation Command”

addDefines

Purpose Add preprocessor macro definitions to model's build information

Syntax `addDefines(buildinfo, macrodefs, groups)`
groups is optional.

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

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

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

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

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

You can apply

- A single group name to an macro definition
- A single group name to multiple macro definitions
- Multiple group names to collections of multiple macro definitions

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

Description

The `addDefines` function adds specified preprocessor macro definitions to the model's build information. The Real-Time Workshop software stores the definitions in a vector. The function adds definitions to the end of the vector based on the order in which you specify them.

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

Examples

- Add the macro definition `-DPRODUCTION` to build information `myModelBuildInfo` and place the definition in the group `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');
```

addDefines

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

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

See Also

`addCompileFlags`, `addLinkFlags`
“Programming a Post Code Generation Command”

Purpose

Add include files to model's build information

Syntax

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

paths and *groups* are optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

filenames

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

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

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

The function removes duplicate include file entries that

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

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

paths (optional)

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

groups (optional)

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

addIncludeFiles

- 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 <i>filenames</i> . |

Description

The `addIncludeFiles` function adds specified include files to the model's build information. The Real-Time Workshop software stores the include files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

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

| If You Specify an Optional Argument as a... | The Function... |
|---|---|
| Character array | Applies the character array to 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 <i>filenames</i> . |

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', '/proj/src', 'SysFiles');
```

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

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

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

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

- Add all of the `.h` files in a specified directory to build information `myModelBuildInfo` and place the files in the group `HFiles`.

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

addIncludeFiles

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

addIncludePaths

| 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 <i>paths</i> . |

Description

The `addIncludePaths` function adds specified include paths to the model's build information. The Real-Time Workshop software stores the include paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

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

| If You Specify an Optional Argument as a... | The Function... |
|---|---|
| Character array | Applies the character array to 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 <i>paths</i> . |

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 <i>options</i> . Available for nonmakefile build environments only. |

Description

The `addLinkFlags` function adds specified linker options to the model's build information. The Real-Time Workshop software stores the linker options in a vector. The function adds options to the end of the vector based on the order in which you specify them.

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

Examples

- Add the linker `-T` option to build information `myModelBuildInfo` and place the option in the group `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');
```

addLinkFlags

- 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 groupTemp.

```
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. If you specify an optional argument, you must specify all of the optional arguments preceding it.

Arguments

buildinfo

Build information returned by RTW.BuildInfo.

linkobjs

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

The function removes duplicate link objects that

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

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

paths

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

addLinkObjects

priority (optional)

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

precompiled (optional)

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

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

If `precompiled` is `false` (the default), the Real-Time Workshop build process creates the link object in the build directory.

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

linkonly (optional)

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

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

If *linkonly* is `false` (the default), rules for building the link objects are added to the makefile. In this case, the value of *precompiled* determines which subsection of the added rules is expanded, `START_PRECOMP_LIBRARIES` (`true`) or `START_EXPAND_LIBRARIES` (`false`).

groups (optional)

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

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

You can apply

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

| To... | Specify groups as a... |
|---|--|
| Apply one group name to 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 <i>linkobjs</i> . |

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

Description

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

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

addLinkObjects

| If You Specify <i>paths</i> or <i>groups</i> 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 <i>linkobjs</i> . |

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

| If You Specify <i>priority</i>, <i>precompiled</i>, or <i>linkonly</i> 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 <i>linkobjs</i> . |

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

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

Examples

- Add the linkable objects `libobj1` and `libobj2` to build information `myModelBuildInfo` and set the priorities of the objects to 26 and 10, respectively. Since `libobj2` is assigned the lower numeric priority

value, and thus has the higher priority, the function orders the objects such that `libobj2` precedes `libobj1` in the vector.

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

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

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

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

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

See Also

“Programming a Post Code Generation Command”

addNonBuildFiles

| | |
|------------------|--|
| Purpose | Add nonbuild-related files to model's build information |
| Syntax | <code>addNonBuildFiles(<i>buildinfo</i>, <i>filenames</i>, <i>paths</i>, <i>groups</i>)</code> <i>paths</i> and <i>groups</i> are optional. |
| Arguments | <p><i>buildinfo</i> Build information returned by <code>RTW.BuildInfo</code>.</p> <p><i>filenames</i> A character array or cell array of character arrays that specifies names of nonbuild-related files to be added to the build information.</p> <p>The filename strings can include wildcard characters, provided that the dot delimiter (.) is present. Examples are <code>'*.*'</code>, <code>'*.DLL'</code>, and <code>'*.D*'</code>.</p> <p>The function adds the filenames to the end of a vector in the order that you specify them.</p> <p>The function removes duplicate nonbuild file entries that</p> <ul style="list-style-type: none">• Already exist in the nonbuild file vector• Have a path that matches the path of a matching filename <p>A duplicate entry consists of an exact match of a path string and corresponding filename.</p> <p><i>paths</i> (optional) A character array or cell array of character arrays that specifies paths to the nonbuild files. The function adds the paths to the end of a vector in the order that you specify them. If you specify a single path as a character array, the function uses that path for all files.</p> <p><i>groups</i> (optional) A character array or cell array of character arrays that groups specified nonbuild files. You can use groups to</p> |

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

You can apply

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

| To... | Specify groups as a... |
|---|---|
| Apply one group name to all nonbuild files | Character array. |
| Apply different group names to nonbuild files | Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>filenames</i> . |

Description

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

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

addNonBuildFiles

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

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

Examples

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

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

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

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

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

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

See Also

`getNonBuildFiles`

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

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

The function removes duplicate source file entries that

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

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

paths (optional)

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

groups (optional)

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

addSourceFiles

- 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 <i>filenames</i> . |

Description

The `addSourceFiles` function adds specified source files to the model's build information. The Real-Time Workshop software stores the source files in a vector. The function adds the filenames to the end of the vector in the order that you specify them.

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

| If You Specify an Optional Argument as a... | The Function... |
|---|--|
| Character array | Applies the character array to 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 <i>filenames</i> . |

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

Examples

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

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

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

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

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

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

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

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

addSourceFiles

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 The Real-Time Workshop software does not check whether a specified path string is valid.

groups (optional)

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

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

addSourcePaths

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 <i>groups</i> 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 <i>paths</i> . |

Description

The `addSourcePaths` function adds specified source paths to the model's build information. The Real-Time Workshop software stores the source paths in a vector. The function adds the paths to the end of the vector in the order that you specify them.

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

| If You Specify an Optional Argument as a... | The Function... |
|---|---|
| Character array | Applies the character array to 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 <i>paths</i> . |

Note The Real-Time Workshop software does not check whether a specified path string is valid.

Examples

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

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

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

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

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

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

See Also

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

addTMFTokens

| | |
|------------------|--|
| Purpose | Add template makefile (TMF) tokens that provide build-time information for makefile generation |
| Syntax | <code>addTMFTokens(<i>buildinfo</i>, <i>tokennames</i>, <i>tokenvalues</i>, <i>groups</i>)</code> <i>groups</i> is optional. |
| Arguments | <p><i>buildinfo</i> Build information returned by <code>RTW.BuildInfo</code>.</p> <p><i>tokennames</i> A character array or cell array of character arrays that specifies names of TMF tokens (for example, ' >CUSTOM_OUTNAME< ') to be added to the build information. The function adds the token names to the end of a vector in the order that you specify them.</p> <p>If you specify a token name that already exists in the vector, the first instance takes precedence and its value used for replacement.</p> <p><i>tokenvalues</i> A character array or cell array of character arrays that specifies TMF token values corresponding to the previously-specified TMF token names. The function adds the token values to the end of a vector in the order that you specify them.</p> <p><i>groups</i> (optional) A character array or cell array of character arrays that groups specified TMF tokens. You can use groups to</p> <ul style="list-style-type: none">• Document the use of specific TMF tokens• Retrieve or apply groups of TMF tokens <p>You can apply</p> <ul style="list-style-type: none">• A single group name to a TMF token• A single group name to multiple TMF tokens• Multiple group names to collections of multiple TMF tokens |

| To... | Specify groups as a... |
|---|--|
| Apply one group name to all TMF tokens | Character array. |
| Apply different group names to TMF tokens | Cell array of character arrays such that the number of group names that you specify matches the number of elements you specify for <i>tokennames</i> . |

Description

Call the `addTMFTokens` function inside a post code generation command to provide build-time information to help customize makefile generation. The tokens specified in the `addTMFTokens` function call must be handled appropriately in the template makefile (TMF) for the target selected for your model. For example, if your post code generation command calls `addTMFTokens` to add a TMF token named `|>CUSTOM_OUTNAME<|` that specifies an output file name for the build, the TMF must take appropriate action with the value of `|>CUSTOM_OUTNAME<|` to achieve the desired result. (See “Example” on page 3-36.)

The `addTMFTokens` function adds specified TMF token names and values to the model’s build information. The Real-Time Workshop software stores the TMF tokens in a vector. The function adds the tokens to the end of the vector in the order that you specify them.

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

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

addTMFTokens

Example

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

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

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

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

See Also

“Programming a Post Code Generation Command”

Purpose

Generate embeddable C code from MATLAB code

Syntax

```
emlc -options files fcn
```

Description

`emlc -options files fcn` translates the MATLAB function in `fcn` `.m` or `fcn.p` to an embeddable C library or executable, or to a MEX function, based on compiler *options*. Optionally, you can specify custom *files* to include in the build.

Inputs

fcn

MATLAB function from which to generate a MEX function, C library, or C executable code. *fcn* must comply with Embedded MATLAB™ syntax and semantics, a subset of the MATLAB language.

files

Space-separated list of custom files to include in generated code. You can include the following types of files:

- C file (.c)
- Header file (.h)
- Object file (.o or .obj)
- Library (.a, .so, or .lib)
- Template makefile (.tmf)

options

Choice of compiler options. `emlc` resolves options from left to right. If you specify conflicting options, the rightmost one prevails.

`-c` Generate C code, but do not invoke the make command. Use only with `rtw`, `rtw:exe`, and `rtw:lib` targets.

`-d out_folder` Store generated files in the absolute or relative path specified by `out_folder`. If the folder specified by `out_folder` does not exist, `emlc` creates it for you.

If not specified, generates files in the default folder:

```
emcprj/target/fcn
```

`target` can be:

- `mexfcn` for MEX functions
- `rtwexe` for embeddable C executables
- `rtwlib` for embeddable C libraries

`fcn` is the name of the MATLAB function.

| | |
|--|--|
| <code>-eg <i>example_inputs</i></code> | Define the size, class, and complexity of all MATLAB function inputs. Use the values in <i>example_inputs</i> to define these properties. <i>example_inputs</i> must be a cell array that specifies the same number and order of inputs as the MATLAB function. |
| <code>-F <i>fimath_obj</i></code> | Specify <i>fimath_obj</i> as the default <i>fimath</i> object for all fixed-point inputs to the primary function. You can define <i>fimath_obj</i> using the <i>fimath</i> function. If not specified, emlc uses the MATLAB default <i>fimath</i> value. |
| <code>-g</code> | Compiles MEX functions in debug mode, with optimization turned off. Use only for mex targets. If not specified, emlc generates MEX functions in optimized mode. |
| <code>-I <i>include_path</i></code> | Add <i>include_path</i> to the beginning of the Embedded MATLAB path. emlc searches the Embedded MATLAB path <i>first</i> when converting M-code to C code. |

`-N numerictype_obj`

Specify *numerictype_obj* as the default *numerictype* object for all fixed-point inputs to the MATLAB function.

You can define *numerictype_obj* using the *numerictype* function.

If not specified, you must define fixed-point inputs using the `-eg` option.

`-o output_file_name`

Generate the MEX function, C library, or C executable file with the base name *output_file_name* plus an extension:

- `.a` or `.lib` for C libraries
- `.exe` or no extension for C executables
- Platform-dependent extension for generated MEX functions

output_file_name can be a file name or include an existing path.

| | |
|-------------------------------------|--|
| <code>-O optimization_option</code> | <p>Optimize generated code, based on the value of <i>optimization_option</i>:</p> <ul style="list-style-type: none">• <code>enable:inline</code> — Enable function inlining• <code>disable:inline</code> — Disable function inlining• <code>enable:blas</code> — Use BLAS library, if available• <code>disable:blas</code> — Do not use BLAS library <p>If not specified, emlc uses inlining and BLAS library functions for optimization.</p> |
| <code>-report</code> | <p>Generate a compilation report. If not specified, emlc generates a report only if there are error or warning messages.</p> |
| <code>-s config_obj</code> | <p>Specify code generation parameters, based on <i>config_obj</i>, defined as one of these objects:</p> <ul style="list-style-type: none">• <code>emlcoder.HardwareImplementation</code> — Parameters of the target hardware for embeddable C code. Use with <code>rtw</code>, <code>rtw:lib</code>, or <code>rtw:exe</code> targets. If not specified, emlc generates code compatible with the MATLAB host computer. |

- `emlcoder.MEXConfig` — Parameters for MEX code generation. Use with `mex` target.
- `emlcoder.RTWConfig` — Parameters for embeddable C code generation. Use with `rtw`, `rtw:lib`, and `rtw:exe` targets.

Define `config_obj` as a MATLAB variable. For example:

```
rtw_config = emlcoder.RTWConfig
```

```
-T target_option
```

Specify target for generated code, based on value of `target_option`:

- `mex` — Generate a MEX function (default)
- `rtw` or `rtw:lib` — Generate embeddable C library file
- `rtw:exe` — Generate embeddable C code executable file. You must provide a `main` function to include in the build.

If not specified, `emlc` generates a MEX function.

| | |
|----|---|
| -v | Enable verbose mode to show compilation steps. Use with <code>rtw</code> , <code>rtw:lib</code> , and <code>rtw:exe</code> targets. |
| -? | Display help for <code>emlc</code> command. |

Examples

Generate a MEX function from an Embedded MATLAB compliant function:

- 1 Write a MATLAB function `emcrand` that generates a random scalar value drawn from the standard uniform distribution on the open interval (0,1):

```
function r = emcrand() %#eml
% The directive %#eml declares the function
% to be Embedded MATLAB compliant
r = rand();
```

- 2 Generate and run the MEX function:

```
emlc emcrand
emcrand
```

Generate C executable files from an Embedded MATLAB compliant function. Specify the main C function as a configuration parameter:

- 1 Write a MATLAB function `emcrand` that generates a random scalar value drawn from the standard uniform distribution on the open interval (0,1):

```
function r = emcrand() %#eml
r = rand();
```

- 2 Write a main C function `c:\myfiles\main.c` that calls `emcrand`:

```
/*
** main.c
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    emcrand_initialize();

    printf("emcrand=%g\n", emcrand());

    emcrand_terminate();

    return 0;
}
```

- 3 Configure your code generation parameters to include the main C function, then generate the C executable:

```
rtwcfg = emlcoder.RTWConfig
rtwcfg.CustomSource = 'main.c'
rtwcfg.CustomInclude = 'c:\myfiles'
emlc -T rtw:exe -s rtwcfg emcrand
```

emlc generates C executables and supporting files in the default folder `emcprj/rtwexe/emcrand`.

This example shows how to specify a main function as a parameter in the configuration object `emlcoder.RTWConfig`. Alternatively, you can specify the file containing `main()` separately on the command line. This file can be a source, object, or library file.

Generate C library files in a custom folder from an Embedded MATLAB compliant function with inputs of different classes and sizes. The first

input is a 1-by-4 vector of unsigned 16-bit integers; the second input is a double-precision scalar:

- 1 Write a MATLAB function `emcadd` that returns the sum of two values:

```
function y = emcadd(u,v) %#eml
y = u + v;
```

- 2 Generate the C library files in a custom folder `emcaddlib`:

```
emlc -T rtw:lib -d emcaddlib -eg {zeros(1,4,'uint16'),0} emcadd
```

Generate C library files from an Embedded MATLAB compliant function that takes a fixed-point input:

- 1 Write an M-function `emcsqrtfi` that computes the square root of a fixed-point input:

```
function y = emcsqrtfi(x) %#eml
y = sqrt(x);
```

- 2 Define `numerictype` and `fimath` properties for the fixed-point input `x` and generate C library code for `emcsqrtfi`:

```
T = numerictype('WordLength',32, ...
               'FractionLength',23, ...
               'Signed',true)
F = fimath('SumMode','SpecifyPrecision', ...
          'SumWordLength',32, ...
          'SumFractionLength',23, ...
          'ProductMode','SpecifyPrecision', ...
          'ProductWordLength',32, ...
          'ProductFractionLength',23)
% Define a fixed-point variable with these
% numerictype and fimath properties
myfiprops = {fi(4.0,T,F)}
emlc -T rtw:lib -eg myfiprops emcsqrtfi
```

emlc generates C library and supporting files in the default folder `emcprj/rtwlib/emcsqrtfi`.

Alternatives

You can use a GUI to modify parameters for code generation with emlc:

- To modify hardware implementation parameters using a dialog box:

```
hw_config = emlcoder.HardwareImplementation
open hw_config
```

- To modify C code generation parameters using a dialog box:

```
rtw_config = emlcoder.RTWConfig
open rtw_config
```

- To modify MEX code generation parameters using a dialog box:

```
mex_config = emlcoder.MEXConfig
open mex_config
```

See Also

`fimath` | `numericitype` | `emlmex` | `mex` | `fi`

Tutorials

- “Generating C Code from MATLAB Code for Embedded Applications”
- Generating Code for Embedded MATLAB

How To

- “Making M-Code Compliant with the Embedded MATLAB Subset”
- “Converting MATLAB Code to C Code”
- “Specifying Properties of Primary Function Inputs”
- “Configuring Your Environment for Code Generation”
- “File Paths and Naming Conventions”

| | |
|--------------------|--|
| Purpose | Find and add include (header) files to build information object |
| Syntax | <code>findIncludeFiles(<i>buildinfo</i>, <i>extPatterns</i>)</code> <i>extPatterns</i> is optional. |
| Arguments | <p><i>buildinfo</i> Build information returned by <code>RTW.BuildInfo</code>.</p> <p><i>extPatterns</i> (optional) A cell array of character arrays that specify patterns of file name extensions for which the function is to search. Each pattern</p> <ul style="list-style-type: none">• Must start with <code>*</code>.• Can include any combination of alphanumeric and underscore (<code>_</code>) characters <p>The default pattern is <code>*.h</code>.</p> <p>Examples of valid patterns include</p> <ul style="list-style-type: none"><code>*.h</code><code>*.hpp</code><code>*.x*</code> |
| Description | <p>The <code>findIncludeFiles</code> function</p> <ul style="list-style-type: none">• 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 |

findIncludeFiles

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”

| | |
|--------------------|---|
| Purpose | Compiler options from model's build information |
| Syntax | <pre>options = getCompileFlags(buildinfo, includeGroups, excludeGroups)</pre> <p><i>includeGroups</i> and <i>excludeGroups</i> are optional.</p> |
| Arguments | <p><i>buildinfo</i> Build information returned by RTW.BuildInfo.</p> <p><i>includeGroups</i> (optional) A character array or cell array of character arrays that specifies groups of compiler flags you want the function to return.</p> <p><i>excludeGroups</i> (optional) A character array or cell array of character arrays that specifies groups of compiler flags you do not want the function to return.</p> |
| Returns | Compiler options stored in the model's build information. |
| Description | <p>The <code>getCompileFlags</code> function returns compiler options stored in the model's build information. Using optional <i>includeGroups</i> and <i>excludeGroups</i> arguments, you can selectively include or exclude groups of options the function returns.</p> <p>If you choose to specify <i>excludeGroups</i> and omit <i>includeGroups</i>, specify a null string ('') for <i>includeGroups</i>.</p> |
| Examples | <ul style="list-style-type: none">Get all compiler options stored in build information <code>myModelBuildInfo</code>. <pre>myModelBuildInfo = RTW.BuildInfo; addCompileFlags(myModelBuildInfo, {'-Zi -Wall' '-O3'},... {'Debug' 'MemOpt'});</pre> |

getCompileFlags

```
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”

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 |
|--------------------|---|
| <i>macrodefs</i> | Complete macro definitions with -D prefix |
| <i>identifiers</i> | Names of the macros |
| <i>values</i> | Values assigned to the macros (anything specified to the right of the first equals sign) ; the default is an empty string (' ') |

getDefines

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'

```

getDefines

See Also

getCompileFlags, getLinkFlags
“Programming a Post Code Generation Command”

Purpose All files from model's build information

Syntax `[fPathNames, names] = getFullFileList(buildinfo, fcase)`
fcase is optional.

Arguments *buildinfo*
 Build information returned by RTW.BuildInfo.

fcase (optional)
 The string 'source', 'include', or 'nonbuild'. If the argument is omitted, the function returns all files from the model's build information.

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

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

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

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

Examples List all the files stored in build information `myModelBuildInfo`.

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

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... |
|--------------------|---|
| <code>true</code> | Concatenates and returns each filename with its corresponding path. |
| <code>false</code> | Returns only filenames. |

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

| If You Specify... | The Function... |
|--------------------|---|
| <code>true</code> | Replaces the token <code>\$(MATLAB_ROOT)</code> with the absolute path string for your MATLAB installation directory. |
| <code>false</code> | Does not replace the token <code>\$(MATLAB_ROOT)</code> . |

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]
```

getIncludeFiles

- 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.

getIncludePaths

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/etcclib'...
'/etcproj/etc/etc_build' '/common/lib'},...
{'etc' 'etc' 'shared'});
incpaths=getIncludePaths(myModelBuildInfo, false);
incpaths
```

```
incpaths =
```

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

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

```
myModelBuildInfo = RTW.BuildInfo;
addIncludePaths(myModelBuildInfo, {'/etc/proj/etcclib'...
'/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 | <pre>options=getLinkFlags(buildinfo, includeGroups, excludeGroups)</pre> <p><i>includeGroups</i> and <i>excludeGroups</i> are optional.</p> |
| Arguments | <p><i>buildinfo</i> Build information returned by RTW.BuildInfo.</p> <p><i>includeGroups</i> (optional) A character array or cell array that specifies groups of linker flags you want the function to return.</p> <p><i>excludeGroups</i> (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 <i>includeGroups</i>.</p> |
| Returns | Linker options stored in the model's build information. |
| Description | <p>The getLinkFlags function returns linker options stored in the model's build information. Using optional <i>includeGroups</i> and <i>excludeGroups</i> arguments, you can selectively include or exclude groups of options the function returns.</p> <p>If you choose to specify <i>excludeGroups</i> and omit <i>includeGroups</i>, specify a null string ('') for <i>includeGroups</i>.</p> |

getLinkFlags

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”

getNonBuildFiles

Purpose Nonbuild-related files from model's build information

Syntax `files=getNonBuildFiles(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 nonbuild files you want the function to return.

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

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

Description The `getNonBuildFiles` function returns the names of nonbuild-related files, such as DLL files required for a final executable, or a README file, stored in the model'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 nonbuild files the function returns.

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

Examples Get all nonbuild filenames stored in build information
`myModelBuildInfo`.

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

nonbuildfiles =

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

See Also `addNonBuildFiles`

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'
```

getSourceFiles

- 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)`
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 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.

getSourcePaths

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', 'src'));
```

```
srcpaths=getSourcePaths(myModelBuildInfo, false);  
srcpaths{:}
```

```
ans =
```

```
$(MATLAB_ROOT)\rtw\c\src
```

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

```
ans =
```

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

See Also

`getIncludeFiles`, `getIncludePaths`, `getSourceFiles`
“Programming a Post Code Generation Command”

packNGo

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: <ul style="list-style-type: none">• <code>m1rFiles.zip</code> — files in your <i>matlabroot</i> directory tree• <code>sDirFiles.zip</code> — files in and under your build directory• <code>otherFiles.zip</code> — required files not in the <i>matlabroot</i> or <i>start</i> 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: ' <i>model.zip</i> ' If you omit the <i>.zip</i> 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 files)
- Header files (for example, .h and .hpp files)
- Nonbuild-related files (for example, .dll files required for a final executable and .txt informational files)
- MAT-file that contains the model's build information object (.mat file)

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

By default, the function packages the files as a flat 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”

rsimgetrtp

Purpose Global model parameter structure

Syntax
`rsimgetrtp('model')`
`rsimgetrtp('model', 'AddTunableParamInfo' 'value')`

Description `rsimgetrtp('model')` forces a block update diagram action for *model*, a model for which you are running rapid simulations, and returns the global parameter structure for that model.

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

The model parameter structure contains the following fields:

| Field | Description |
|----------------------------|--|
| <code>modelChecksum</code> | A four-element vector that encodes the structure. The Real-Time Workshop software uses the <i>checksum</i> to check whether the structure has changed since the RSim executable was generated. If you delete or add a block, and then generate a new version of the structure, the new <i>checksum</i> will not match the original <i>checksum</i> . The RSim executable detects this incompatibility in model parameter structures and exits to avoid returning incorrect simulation results. If the structure changes, you must regenerate code for the model. |
| <code>parameters</code> | A structure that defines model global parameters. |

The `parameters` substructure includes the following fields:

| Field | Description |
|--------------|---|
| dataTypeName | Name of the parameter data type, for example, <code>double</code> |
| dataTypeID | An internal data type identifier |
| complex | Value 1 if parameter values are complex and 0 if real |
| dtTransIdx | Internal use only |
| values | Vector of parameter values |

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

| Field | Description |
|--------------|--|
| Identifier | Parameter name |
| ValueIndices | Vector of indices to parameter values |
| Dimensions | Vector indicating parameter dimensions |

Examples

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

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

param_struct =

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

See Also

`rsimsetrtpparam`

How To

- “Creating a MAT-File That Includes a Model Parameter Structure”

- “Updating a Block Diagram”
- “Inline parameters”
- “Implementing Block Features”
- “Tuning Parameters”

Purpose

Set parameters of rtP model parameter structure

Syntax

```
rtp = rsimsetrtpparam(rtp, idx)
rtp = rsimsetrtpparam(rtp, 'paramName', paramValue)
rtP = rsimsetrtpparam( rtP, idx, 'paramName', paramValue )
```

Description

```
rtp = rsimsetrtpparam(rtp, idx)
```

Expands the rtP structure to have idx sets of parameters

```
rtp = rsimsetrtpparam(rtp, 'paramName', paramValue)
```

Takes an rtP structure with tunable parameter information and sets the values associated with 'paramName' to be paramValue if possible. There can be more than one name-value pair.

```
rtP = rsimsetrtpparam( rtP, idx, 'paramName', paramValue )
```

The rsimsetrtpparam utility allows for defining the values of an existing rtP parameter structure.

Takes an rtP structure with tunable parameter information and sets the values associated with 'paramName' to be paramValue in the idx'th parameter set. There can be more than one name-value pair. If the rtP structure does not have idx parameter sets, the first set is copied and appended until there are idx parameter sets. Subsequently, the idx'th set is changed.

Inputs

rtP

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

idx

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

paramValue

The value of the rtP parameter, paramName

rsimsetrtpparam

paramName

The name of the parameter set to add to the rtp structure

Outputs

rtp

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

Definitions

The rtp structure should match the format of the structure returned by rsimsetrtpparam(modelName).

Examples

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

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

2 Add three parameter sets to the parameter structure, 'rtp'.

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

See Also

rsimgetrtpparam

Purpose

Build libraries within model without building model

Syntax

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

Description

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

Inputs

model

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

`build_spec`

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

Field

Value

`rtwmakecfgDirs` Cell array of strings that names the folders containing `rtwmakecfg` files for libraries that you want to precompile. Uses the `Name` and `Location` elements of `makeInfo.library`, as returned by the `rtwmakecfg` function, to specify name and location of precompiled libraries. If you use the `TargetPreCompLibLocation` parameter to specify the library folder, it overrides the `makeInfo.library.Location` setting.

The specified model must contain blocks that use precompiled libraries that the `rtwmakecfg` files specify. The template makefile (TMF)-to-makefile conversion generates the library rules only if the conversion needs the libraries.

rtw_precompile_libs

| Field | Value |
|----------------------------|---|
| libSuffix (optional) | String that specifies the suffix, including the file type extension, to append to the name of each library (for example, <code>.a</code> or <code>_vc.lib</code>). The string must include a period (<code>.</code>). Set the suffix with either this field or the <code>TargetLibSuffix</code> parameter. If you specify a suffix with both mechanisms, the <code>TargetLibSuffix</code> setting overrides the setting of this field. |
| intOnlyBuild (optional) | Boolean flag. When set to <code>true</code> , indicates the function optimizes the libraries so that they compile from integer code only. Applies to ERT-based targets only. |
| makeOpts (optional) | String that specifies an option to include in the <code>rtwMake</code> command line. |
| addLibs (optional) | Cell array of structures that specify the libraries to build that an <code>rtwmakecfg</code> function does not specify. Define each structure with two fields that are character arrays: <ul style="list-style-type: none">• <code>libName</code> — name of the library without a suffix• <code>libLoc</code> — location for the precompiled library The TMF can specify other libraries and how to build them. Use this field if you must precompile libraries. |

Examples

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

```
% Specify the library suffix
if isunix
    suffix = '.a';
```

```
else
    suffix = '_vc.lib';
end
set_param(my_model, 'TargetLibSuffix', suffix);

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

% Define a build specification that specifies the location of the files to compile.
build_spec = [];
build_spec.rtwmakecfgDirs = {fullfile(pwd, 'src')};

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

How To

- “Precompiling S-Function Libraries”
- “Recompiling Precompiled Libraries”

Purpose

Initiate build process

Syntax

```
rtwbuild (model)  
rtwbuild (model, 'OkayToPushNags', true)  
blockHandle = rtwbuild('subsystem')  
blockHandle = rtwbuild('subsystem', 'Mode',  
    'ExportFunctionCalls')
```

Description

`rtwbuild (model)` initiates the build process for the specified model using the current model configuration settings. The argument is a handle to the model or a string specifying the model name. `rtwbuild` creates an executable if you clear the **Generate code only** check box in the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

`rtwbuild (model, 'OkayToPushNags', true)` initiates the build process. Specify the parameter `OkayToPushNags` with the value `true` if you want `rtwbuild` to display any build errors that occur in the Simulink Diagnostic Viewer as well as in the MATLAB command window. If the parameter is omitted or set to `false`, build errors are displayed only in the MATLAB command window.

`blockHandle = rtwbuild('subsystem')` initiates the build process for the specified subsystem using the current model configuration settings. The argument is a string specifying the subsystem name or the full block path for that subsystem (for example, `'rtwdemo_export_functions/rtwdemo_subsystem'`). If you are licensed for Real-Time Workshop Embedded Coder software and select the **Create Simulink (S-Function) block** check box in the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box, `rtwbuild` returns a nonempty block handle, `blockHandle`, to an automatically generated ERT S-function wrapper for the subsystem code.

`blockHandle = rtwbuild('subsystem', 'Mode', 'ExportFunctionCalls')` initiates the build process to export function calls from the specified subsystem. You must be

licensed for Real-Time Workshop Embedded Coder software to export function-call subsystems.

If the model or subsystem is not loaded into the MATLAB environment, `rtwbuild` loads it before initiating the build process.

Examples

Build the `rtwdemo_f14` demo model:

```
rtwbuild('rtwdemo_f14')
```

Build the `rtwdemo_subsystem` function-call subsystem inside the `rtwdemo_export_functions` demo model:

```
rtwdemo_export_functions  
rtwbuild('rtwdemo_export_functions/rtwdemo_subsystem','Mode','ExportFunctionCalls')
```

Alternatives

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

- Clear the **Generate code only** option on the **Real-Time Workshop** pane of the Configuration Parameters dialog box and click **Build**.
- Press **Ctrl+B**.
- Select **Tools > Real-Time Workshop > Build Model**.
- Invoke the `slbuild` command from the MATLAB command line. (For more information on `slbuild`, see [Initiating the Build Process](#).)

How To

- [Initiating the Build Process](#)
- [“Building Executables”](#)
- [Automatic S-Function Wrapper Generation](#)
- [Exporting Function-Call Subsystems](#)

RTW.getBuildDir

Purpose Build directory information for specified model

Syntax `struct=RTW.getBuildDir(modelName)`

Arguments *modelName*
String specifying the name of a Simulink model, which can be open or closed.

Returns Structure containing the following build directory information about the specified model:

| Field | Description |
|--------------------------|---|
| BuildDirectory | String specifying the fully qualified path to the build directory for the model. |
| RelativeBuildDir | String specifying the build directory relative to the current working directory (pwd). |
| BuildDirSuffix | String specifying the suffix appended to the model name to create the build directory. |
| ModelRefRelativeBuildDir | String specifying the model reference target build directory relative to current working directory (pwd). |
| ModelRefRelativeSimDir | String specifying the model reference target simulation directory relative to current working directory (pwd). |
| ModelRefDirSuffix | String specifying the suffix appended to the system target file name to create the model reference build directory. |

Description The `RTW.getBuildDir` function returns build directory information for a specified model, which can be open or closed. If the model is closed, the function opens and then closes the model, leaving it in its original state.

This function can be used in automated scripts to programmatically determine the build directory in which a model's generated code would be placed if the model were built in its current state.

Note The `RTW.getBuildDir` function may take significantly longer to execute if the specified model is large and closed.

Example

Return build directory information for the model `mymodel`.

```
>> info=RTW.getBuildDir('mymodel');  
>> info
```

```
info =
```

```
      BuildDirectory: 'c:\work\mymodel_ert_rtw'  
      RelativeBuildDir: 'mymodel_ert_rtw'  
      BuildDirSuffix: '_ert_rtw'  
ModelRefRelativeBuildDir: 'slprj\ert\mymodel'  
ModelRefRelativeSimDir: 'slprj\sim\mymodel'  
ModelRefDirSuffix: ''
```

rtwrebuild

Purpose Rebuild generated code

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

Description `rtwrebuild()` recompiles files you modified by invoking the makefile generated during the previous build. If there are no inputs, your current working directory is the build directory of the model.

Use `rtwrebuild('model')` if your current working directory is one level above the build directory of the model (pwd when you initiated the Real-Time Workshop build).

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

If your model includes submodels, the Real-Time Workshop software builds the submodels recursively before rebuilding the top model.

| | | |
|---------------|--------------|--|
| Inputs | <i>model</i> | String specifying the model name. |
| | <i>path</i> | String specifying the fully qualified path to the build directory for the model. |

Examples Rebuild the `rtwdemo_f14` model:

```
rtwrebuild('rtwdemo_f14')
```

Rebuild the model in a specified path:

```
rtwrebuild(fullfile(matlabroot,'rtwdemo_f14'))
```

See Also “Rebuilding a Model” — Describes rebuilding the generated code from the Model Explorer.

| | |
|---------------------|--|
| Purpose | Generate report documenting the generated code for a model |
| Syntax | <pre>rtwreport(model) rtwreport(model, directory)</pre> |
| Description | <p><code>rtwreport(model)</code> generates a report that shows the generated code for a model. <i>model</i> is a string enclosed in quotes specifying the model name. If necessary, the function loads the model and generates code before generating the report. The report includes:</p> <ul style="list-style-type: none">• Snapshots of block diagrams of the model and its subsystems• Block execution order list• Code generation summary that includes a list of generated code files, configuration settings, a subsystem map, and a traceability report.• Full listings of generated code that resides in the build directory <p>By default, the Real-Time Workshop software names the generated report <code>codegen.html</code> and places the file in your current directory.</p> <p><code>rtwreport(model, directory)</code> includes a user-specified directory. <i>directory</i> is a string enclosed in quotes specifying the name of a directory. The Real-Time Workshop software places the generated report in the parent directory of the directory you specify. The Real-Time Workshop project directory (<code>s1prj</code>) must be in the parent directory. If the user-specified directory cannot be found, an error results and the Real-Time Workshop software does not generate code.</p> |
| Examples | <p>Generate a report for model <code>rtwdemo_codegenrpt</code>:</p> <pre>rtwreport('rtwdemo_codegenrpt');</pre> |
| Alternatives | In the model window, select Tools > Report Generator . See <i>Simulink® Report Generator™ User's Guide</i> for more information. |
| Tutorials | <ul style="list-style-type: none">• “Documenting a Code Generation Project” |

How To

- “What Is the Report Explorer?”
- Code Generation Summary
- Import Generated Code

Purpose Trace block to generated code

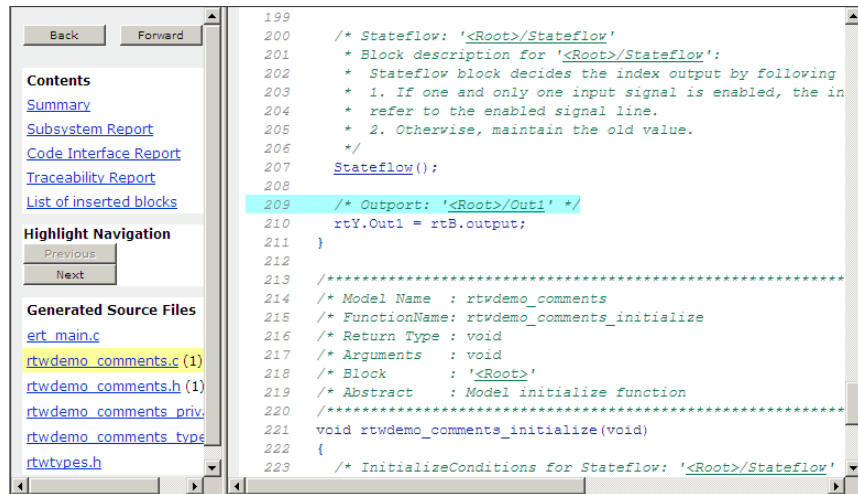
Syntax `rtwtrace(blockpath)`

Description `rtwtrace(blockpath)` opens an HTML code generation report, that displays contents of the source code file, and highlights the line of code corresponding to the specified block. *blockpath* is a string enclosed in quotes specifying the full Simulink block path, for example, *'model_name/block_name'*. Before calling `rtwtrace`, you must enable model to code navigation. On the Configuration parameter dialog box, select **Real-Time Workshop > Report** pane, and select the **Model-to-code** parameter. Generate code for your model using the Real-Time Workshop Embedded Coder software. The build directory must be under the current working directory, otherwise `rtwtrace` might produce an error.

Examples Trace to the source code for block Out1 in `rtwdemo_comments.mdl`:

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

The HTML code generation report opens and highlights the first instance of code generated for block Out1.



Alternatives

To trace from a block in the model diagram, right-click a block and select **Real-Time Workshop > Navigate to code**.

How To

- “Tracing Blocks to Generated Code”
- “Model-to-code” on page 6-47

Purpose

Specify target for configuration set

Syntax

```
switchTarget('config_set', 'sys_tgt_file', target_options)
```

Description

switchTarget('config_set', 'sys_tgt_file', target_options) specifies a system target file for the configuration set that you specify.

Inputs

config_set

Handle to the active configuration set for the model.

sys_tgt_file

String that specifies a system target file.

target_options

Structure of field and value pairs to optionally specify the template makefile, TLC options, make command, and description associated with the target. If you do not want to use any options, you must specify an empty structure ([]).

| Field | Value |
|------------------|---|
| TemplateMakefile | String specifying file name of template makefile. |
| TLCOptions | String specifying TLC argument. |
| MakeCommand | String specifying make command M-file. |
| Description | String specifying a description of the target. |

Examples

Select an ert.tlc system target file for the active configuration set:

```
% Get the active configuration set for 'model'
cs = getActiveConfigSet(model);
% Define a system target file
stf = 'ert.tlc';
```

switchTarget

```
% Change the system target file for the configuration set.
switchTarget(cs,stf,[]);
```

Specify an ert.tlc system target file and target options for the active configuration set:

```
% Get the active configuration set for 'model'
cs = getActiveConfigSet(model);
% Define a system target file
stf = 'ert.tlc';
% Specify target options
tgtOpt.TemplateMakefile = 'grt_default_tmf';
tgtOpt.TLCOptions = '-aVarName=1';
tgtOpt.MakeCommand = 'make_rtw';
tgtOpt.Description = 'my target';
% Change the system target file and target options
% for the configuration set.
switchTarget(cs,stf,tgtOpt);
```

Alternatives

To select system target files using the Configuration Parameters dialog box:

- 1** In your model, open the Configuration Parameters dialog box.
- 2** Navigate to the **Real-Time Workshop > General** pane.
- 3** Specify the **System target file**.
- 4** Optionally specify , **Make command**, and **TLC options**.
- 5** Click **Apply**.

How To

- “Selecting a System Target File Programmatically”
- “Selecting a Target ”
- “Setting Target Language Compiler Options”

Purpose Invokes Target Language Compiler to convert model description file to generated code.

Syntax `tlc [-options] [file]`

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

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

`tlc [-options] [file]`

You can change the default behavior by specifying one or more compilation *options* as described in “Options” on page 3-93

Options

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

- “-r Specify Real-Time Workshop filename” on page 3-94
- “-v Specify verbose level” on page 3-94
- “-l Specify path to local include files” on page 3-94
- “-m Specify maximum number of errors” on page 3-94
- “-O Specify the output file path” on page 3-94

- “-d[a|c|n|o] Invoke debug mode” on page 3-95
- “-a Specify parameters” on page 3-95
- “-p Print progress” on page 3-95
- “-lint Performance checks and runtime statistics” on page 3-95
- “-xO Parse only” on page 3-95

-r Specify Real-Time Workshop filename

-r file_name

Specify the filename that you want to translate.

-v Specify verbose level

-v number

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

-l Specify path to local include files

-l path

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

-m Specify maximum number of errors

-m number

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

If you omit this option, the default value is five.

-O Specify the output file path

-O path

Specify the directory path to place output files.

If you omit this option, TLC places output files in the current directory.

-d[a|c|n|o] Invoke debug mode

- da execute any %assert directives
- dc invoke the TLC command line debugger
- dn produce log files, which indicate those lines hit and those lines missed during compilation.
- do disable debugging behavior

-a Specify parameters

-a *identifier = expression*

Specify parameters to change the behavior of your TLC program. For example, this option is used by the Real-Time Workshop software to set inlining of parameters or file size limits.

-p Print progress

-p *number*

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

-lint Performance checks and runtime statistics

-lint

Perform simple performance checks and collect runtime statistics.

-xO Parse only

-xO

Parse only a TLC file; do not execute it.

updateFilePathsAndExtensions

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
```

updateFilePathsAndExtensions

See Also

addIncludeFiles, addIncludePaths, addSourceFiles,
addSourcePaths, updateFileSeparator
“Programming a Post Code Generation Command”

| | |
|--------------------|--|
| Purpose | Change file separator used in model's build information |
| Syntax | <code>updateFileSeparator(<i>buildinfo</i>, <i>separator</i>)</code> |
| Arguments | <p><i>buildinfo</i> Build information returned by <code>RTW.BuildInfo</code>.</p> <p><i>separator</i> A character array that specifies the file separator <code>\</code> (Windows®) or <code>/</code> (UNIX®) to be applied to all file path specifications.</p> |
| Description | <p>The <code>updateFileSeparator</code> function changes all instances of the current file separator (<code>/</code> or <code>\</code>) in a model's build information to the specified file separator.</p> <p>The default value for the file separator matches the value returned by the MATLAB command <code>filesep</code>. For makefile based builds, you can override the default by defining a separator with the <code>MAKEFILE_FILESEP</code> macro in the template makefile (see “Cross-Compiling Code Generated on a Microsoft® Windows System”). If the <code>GenerateMakefile</code> parameter is set, the Real-Time Workshop software overrides the default separator and updates the model's build information after evaluating the <code>PostCodeGenCommand</code> configuration parameter.</p> |
| Examples | <p>Update object <code>myModelBuildInfo</code> to apply the Windows file separator.</p> <pre>myModelBuildInfo = RTW.BuildInfo; updateFileSeparator(myModelBuildInfo, '\\');</pre> |
| See Also | <code>addIncludeFiles</code> , <code>addIncludePaths</code> , <code>addSourceFiles</code> , <code>addSourcePaths</code> , <code>updateFilePathsAndExtensions</code> “Programming a Post Code Generation Command”, “Cross-Compiling Code Generated on a Microsoft Windows System” |

updateFileSeparator

Block Reference

Custom Code (p. 4-2)

Insert custom code into generated model files and subsystem functions

Interrupt Templates (p. 4-3)

Create blocks that provide interrupt support for real-time operating system (RTOS)

S-Function Target (p. 4-4)

Generate code for S-function

VxWorks (p. 4-5)

Support VxWorks® applications

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

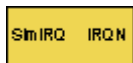
Blocks — Alphabetical List

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. The Stateflow software passes the offsets to the VxWorks call `intConnect(INUM_TO_IVEC(offset), ...)`.

Simulink task priority(s)

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

The **Simulink task priority** values are required to generate 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 The Simulink software does not simulate asynchronous task behavior. The task priority of an asynchronous task is for code generation purposes only and is not honored during simulation.

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

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

Higher priority interrupts can preempt lower priority interrupts in VxWorks. To lock out interrupts during the execution of an ISR, set the preemption flag to 0. This causes generation of `intLock()` and `intUnlock()` calls at the beginning and end of the ISR code. Use interrupt locking carefully, as it increases the system’s interrupt response time for 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.

Async Interrupt

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, the Real-Time Workshop software determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

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

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

Enable simulation input

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

Note Before generating code, consider removing blocks that drive the simulation input to 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
“Handling Asynchronous Events” 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 the code generator creates for the model that contains the block.

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

Parameters **Top of Model Header**
Code to be added 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
"Integrating External Code Using Custom Code Blocks" 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 <i>model.c</i> or <i>model.cpp</i> file that the code generator creates for the model that contains the block. |

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

| | |
|-------------------|---|
| Parameters | Top of Model Source Code to be added 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 "Integrating External Code Using Custom Code Blocks" 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 | <p>An instance of the RTW S-Function block represents code the Real-Time Workshop software 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</p> <ul style="list-style-type: none">• Converting models and subsystems to application components• Reusing models and subsystems• Optimizing simulation — often, an S-function simulates more efficiently than the original model <p>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.</p> |
| Requirements | <ul style="list-style-type: none">• 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). |

RTW S-Function

Parameters

Generated S-function name (`model_sf`)

The name of the generated S-function. The Real-Time Workshop software derives the name by appending `_sf` to the name of the model or subsystem from which the block is generated.

Show module list

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

See Also

“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 <code>SystemDerivatives</code> function that the code generator creates for the model or subsystem that contains the block. |

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

| | |
|-------------------|--|
| Parameters | System Derivatives Function Declaration Code Code to be added to the declaration section of the generated <code>SystemDerivatives</code> function. |
| | System Derivatives Function Execution Code Code to be added to the execution section of the generated <code>SystemDerivatives</code> function. |
| | System Derivatives Function Exit Code Code to be added to the exit section of the generated <code>SystemDerivatives</code> 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
“Integrating External Code Using Custom Code Blocks” 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 the code generator creates for the model or subsystem that contains the block.

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

Parameters **System Disable Function Declaration Code**
Code to be added to the declaration section of the generated `SystemDisable` function.

System Disable Function Execution Code
Code to be added to the execution section of the generated `SystemDisable` function.

System Disable Function Exit Code
Code to be added to the exit section of the generated `SystemDisable` function.

Example See “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
“Integrating External Code Using Custom Code Blocks” 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 <code>SystemEnable</code> function that the code generator creates for the model or subsystem that contains the block. |

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

| | |
|-------------------|--|
| Parameters | System Enable Function Declaration Code Code to be added to the declaration section of the generated <code>SystemEnable</code> function. |
| | System Enable Function Execution Code Code to be added to the execution section of the generated <code>SystemEnable</code> function. |
| | System Enable Function Exit Code Code to be added to the exit section of the generated <code>SystemEnable</code> 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
“Integrating External Code Using Custom Code Blocks” 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 the code generator creates for the model or subsystem that contains the block.

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

Parameters **System Initialize Function Declaration Code**
Code to be added to the declaration section of the generated `SystemInitialize` function.

System Initialize Function Execution Code
Code to be added to the execution section of the generated `SystemInitialize` function.

System Initialize Function Exit Code
Code to be added to the exit section of the generated `SystemInitialize` function.

Example See “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
“Integrating External Code Using Custom Code Blocks” 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 <code>SystemOutputs</code> function that the code generator creates for the model or subsystem that contains the block. |

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

| | |
|-------------------|--|
| Parameters | System Outputs Function Declaration Code Code to be added to the declaration section of the generated <code>SystemOutputs</code> function. |
| | System Outputs Function Execution Code Code to be added to the execution section of the generated <code>SystemOutputs</code> function. |
| | System Outputs Function Exit Code Code to be added to the exit section of the generated <code>SystemOutputs</code> 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
“Integrating External Code Using Custom Code Blocks” 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 the code generator creates for the model or subsystem that contains the block.

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

Parameters **System Start Function Declaration Code**
Code to be added to the declaration section of the generated `SystemStart` function.

System Start Function Execution Code
Code to be added to the execution section of the generated `SystemStart` function.

System Start Function Exit Code
Code to be added to the exit section of the generated `SystemStart` function.

Example See “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
“Integrating External Code Using Custom Code Blocks” 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 <code>SystemTerminate</code> function that the code generator creates for the model or subsystem that contains the block. |

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

| | |
|-------------------|--|
| Parameters | System Terminate Function Declaration Code Code to be added to the declaration section of the generated <code>SystemTerminate</code> function. |
| | System Terminate Function Execution Code Code to be added to the execution section of the generated <code>SystemTerminate</code> function. |
| | System Terminate Function Exit Code Code to be added to the exit section of the generated <code>SystemTerminate</code> 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
“Integrating External Code Using Custom Code Blocks” 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 the code generator creates for the model or subsystem that contains the block.

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

Parameters **System Update Function Declaration Code**
Code to be added to the declaration section of the generated `SystemUpdate` function.

System Update Function Execution Code
Code to be added to the execution section of the generated `SystemUpdate` function.

System Update Function Exit Code
Code to be added to the exit section of the generated `SystemUpdate` function.

Example See “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
“Integrating External Code Using Custom Code Blocks” 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 | <p>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.</p> <p>The Task Sync block performs the following functions:</p> <ul style="list-style-type: none">• Uses the VxWorks system call <code>taskSpawn</code> to spawn an independent task. When the task is activated, it calls the downstream function-call subsystem code or Stateflow chart. The block calls <code>taskDelete</code> 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 <code>for</code> loop. In the loop, the spawned task listens for the semaphore, using <code>semTake</code>. The first call to <code>semTake</code> specifies <code>NO_WAIT</code>. This allows the task to determine whether a second <code>semGive</code> 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, <code>semGive()</code>). 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 |

Task Sync

the Async Interrupt block, or by an independent timer maintained by the task associated with the Task Sync block.

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

Parameters

Task name (10 characters or less)

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

Simulink task priority (0–255)

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

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

Stack size (bytes)

Maximum size to which the task's stack can grow. The stack size is allocated when VxWorks spawns the task. Choose a stack size based on the number of local variables in the task. You should

determine the size by examining the generated code for the task (and 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, the Real-Time Workshop software determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

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

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

Inputs and Outputs

Input

A call from an Async Interrupt block.

Output

A call to a function-call subsystem.

See Also

Async Interrupt

“Handling Asynchronous Events” 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 SimulinkVxWorks Reference. |

Unprotected RT

Configuration Parameters for Simulink Models

- “Real-Time Workshop Pane: General” on page 6-2
- “Real-Time Workshop Pane: Report” on page 6-37
- “Real-Time Workshop Pane: Comments” on page 6-58
- “Real-Time Workshop Pane: Symbols” on page 6-77
- “Real-Time Workshop Pane: Custom Code” on page 6-116
- “Real-Time Workshop Pane: Debug” on page 6-132
- “Real-Time Workshop Pane: Interface” on page 6-141
- “Real-Time Workshop Pane: RSim Target” on page 6-221
- “Real-Time Workshop Pane: Real-Time Workshop S-Function Code Generation Options” on page 6-227
- “Real-Time Workshop Pane: Tornado Target” on page 6-233
- “Parameter Reference” on page 6-261

Real-Time Workshop Pane: General

When the Real-Time Workshop product is installed on your system and you select a GRT-based target, the Real-Time Workshop General pane includes the following parameters.

The image shows a software configuration window titled "Real-Time Workshop Pane: General". It is divided into three main sections:

- Target selection:** Contains a text field for "System target file:" with the value "grt.tlc" and a "Browse..." button. Below it is a dropdown menu for "Language:" set to "C".
- Build process:** Contains a dropdown menu for "Compiler optimization level:" set to "Optimizations off (faster builds)". Below it is a text field for "TLC options:" which is currently empty.
- Makefile configuration:** Contains a checked checkbox for "Generate makefile". Below it are text fields for "Make command:" (value: "make_rtw") and "Template makefile:" (value: "grt_default_tmf").

At the bottom of the window, there are additional controls:

- A dropdown menu for "Select objective:" set to "Unspecified".
- A dropdown menu for "Check model before generating code:" set to "Off", with a "Check model ..." button to its right.
- An unchecked checkbox for "Generate code only".
- A "Build" button.

When the Real-Time Workshop product is installed on your system and you select an ERT-based target, the Real-Time Workshop General pane includes the following parameters. ERT-based target parameters require a Real-Time Workshop Embedded Coder license when generating code.

Target selection

System target file:

Language:

Description: Real-Time Workshop Embedded Coder

Build process

Compiler optimization level:

TLC options:

Makefile configuration

Generate makefile

Make command:

Template makefile:

Data specification override

Ignore custom storage classes Ignore test point signals

Prioritized objectives: Unspecified

Check model before generating code:

Generate code only

In this section...

“Real-Time Workshop: General Tab Overview” on page 6-5

“System target file” on page 6-6

“Language” on page 6-8

“Compiler optimization level” on page 6-10

“Custom compiler optimization flags” on page 6-12

“TLC options” on page 6-13

“Generate makefile” on page 6-15

“Make command” on page 6-17

“Template makefile” on page 6-19

“Ignore custom storage classes” on page 6-21

“Ignore test point signals” on page 6-23

“Select objective” on page 6-25

“Prioritized objectives” on page 6-27

“Set objectives” on page 6-28

“Configuration Set Objectives Dialog Box” on page 6-29

“Check model” on page 6-31

“Check model before generating code” on page 6-32

“Generate code only” on page 6-34

“Build/Generate code” on page 6-36

Real-Time Workshop: General Tab Overview

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

See Also

Real-Time Workshop Pane

System target file

Specify the system target file.

Settings

Default: `grt.tlc`

You can specify the system target file in these ways:

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

Tips

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

Command-Line Information

Parameter: `SystemTargetFile`

Type: `string`

Value: any valid system target file

Default: `'grt.tlc'`

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | No impact |
| Traceability | No impact |

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

See Also

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

Language

Specify C or C++ code generation.

Settings

Default: C

C

Generates .c files and places the files in your build directory.

C++

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

C++ (Encapsulated)

Generates C++ encapsulated .cpp files and places the files in your build directory. Selecting this value causes the build to generate a C++ class interface to model code. The generated interface encapsulates all required model data into C++ class attributes and all model entry point functions into C++ class methods.

Note Using C++ (Encapsulated) for code generation requires a Real-Time Workshop Embedded Coder license and the ERT target. The value C++ (Encapsulated) appears in the **Language** menu if you select an ERT target for your model, but you cannot use the ERT target and the C++ (Encapsulated) value for model building without a Real-Time Workshop Embedded Coder license.

Tip

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

Command-Line Information

Parameter: TargetLang

Type: string

Value: 'C' | 'C++' | 'C++ (Encapsulated)'

Default: 'C'

Recommended Settings

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

See Also

Choosing and Configuring a Compiler

“Controlling Generation of Function Prototypes”

“Controlling Generation of Encapsulated C++ Model Interfaces”

Compiler optimization level

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

Settings

Default: Optimizations off (faster builds)

Optimizations off (faster builds)

Customizes compilation during the Real-Time Workshop makefile build process to minimize compilation time.

Optimizations on (faster runs)

Customizes compilation during the Real-Time Workshop makefile build process to minimize run time.

Custom

Allows you to specify custom compiler optimization flags to be applied during the Real-Time Workshop makefile build process.

Tips

- Target-independent values **Optimizations on (faster runs)** and **Optimizations off (faster builds)** allow you to easily toggle compiler optimizations on and off during code development.
- **Custom** allows you to enter custom compiler optimization flags at Simulink GUI level, rather than editing compiler flags into template makefiles (TMFs) or supplying compiler flags to Real-Time Workshop make commands.
- If you specify compiler options for your Real-Time Workshop makefile build using `OPT_OPTS`, `MEX_OPTS` (except `MEX_OPTS=" -v"`), or `MEX_OPT_FILE`, the value of **Compiler optimization level** is ignored and a warning is issued about the ignored parameter.

Dependencies

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

Command-Line Information

Parameter: RTWCompilerOptimization

Type: string

Value: 'Off' | 'On' | 'Custom'

Default: 'Off'

Recommended Settings

| Application | Setting |
|-------------------|-----------------------------------|
| Debugging | Optimizations off (faster builds) |
| Traceability | Optimizations off (faster builds) |
| Efficiency | Optimizations on (faster runs) |
| Safety precaution | No impact |

See Also

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

Custom compiler optimization flags

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

Settings

Default: ''

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

Dependency

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

Command-Line Information

Parameter: RTWCustomCompilerOptimizations

Type: string

Value: "" | user-specified flags

Default: ""

Recommended Settings

See **Compiler optimization level**.

See Also

- [Compiler optimization level](#)
- [Controlling Compiler Optimization Level and Specifying Custom Optimization Settings](#)

TLC options

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

Settings

Default: ''

You can enter TLC command-line options and arguments.

Tips

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

Command-Line Information

Parameter: TLCOptions

Type: string

Value: any valid TLC argument

Default: ''

Recommended Settings

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

See Also

- TLC Options
- Command-Line Arguments

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

Generate makefile

Specify generation of a makefile.

Settings

Default: on



On

Generates a makefile for a model during the build process.



Off

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

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: GenerateMakefile

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

See Also

- [Customizing Post Code Generation Build Processing](#)
- [Customizing the Target Build Process with the STF_make_rtw Hook File](#)
- [Understanding and Using the Build Process](#)

Make command

Specify a make command and optionally append make command arguments.

Settings

Default: `make_rtw`

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

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

```
make_rtw OPTS=" -DMYDEFINE=1 "
```

The syntax for make command options differs slightly for different compilers.

Tip

Most targets use the default command.

Dependency

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

Command-Line Information

Parameter: MakeCommand

Type: string

Value: any valid make command M-file

Default: 'make_rtw'

Recommended Settings

| Application | Setting |
|-------------------|-----------|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | make_rtw |

See Also

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

Template makefile

Specify a template makefile.

Settings

Default: grt_default_tmf

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

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

Tips

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

Dependency

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

Command-Line Information

Parameter: TemplateMakefile

Type: string

Value: any valid template makefile filename

Default: 'grt_default_tmf'

Recommended Settings

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

See Also

- Template Makefiles and Make Options
- Available Targets

Ignore custom storage classes

Specify whether to apply or ignore custom storage classes.

Settings

Default: off



On

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



Off

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

Tips

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

Dependencies

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

Command-Line Information

Parameter: IgnoreCustomStorageClasses

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

Custom Storage Classes

Ignore test point signals

Specify allocation of memory buffers for test points.

Settings

Default: Off



On

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



Off

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: IgnoreTestpoints

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | Off |
| Traceability | No impact |

| Application | Setting |
|--------------------|----------------|
| Efficiency | On |
| Safety precaution | No impact |

See Also

- “Signals with Test Points” in the Real-Time Workshop User’s Guide
- “Working with Test Points” in the Simulink User’s Guide
- “Signal Considerations” in the Real-Time Workshop User’s Guide

Select objective

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

Settings

Default: Unspecified

Unspecified

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

Debugging

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

Tips

For more objectives, specify an ERT-based target.

Dependency

This parameter appears only for GRT-based targets.

Command-Line Information

Parameter: 'ObjectivePriorities'

Type: cell array of strings

Value: { '' } | { 'Debugging' }

Default: { '' }

Recommended Settings

| Application | Setting |
|-------------------|--------------------------------------|
| Debugging | Debugging |
| Traceability | Not applicable for GRT-based targets |
| Efficiency | Not applicable for GRT-based targets |
| Safety precaution | Not applicable for GRT-based targets |

See Also

- “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.
- “Configuring Code Generation Objectives” in the Real-Time Workshop User’s Guide.

Prioritized objectives

Lists objectives that you specify by clicking the **Set objectives** button.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Command: `get_param('model', 'ObjectivePriorities')`

See Also

- “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.
- “Configuring Code Generation Objectives” in the Real-Time Workshop User’s Guide.

Set objectives

Opens Configuration Set Objectives dialog box.

Dependency

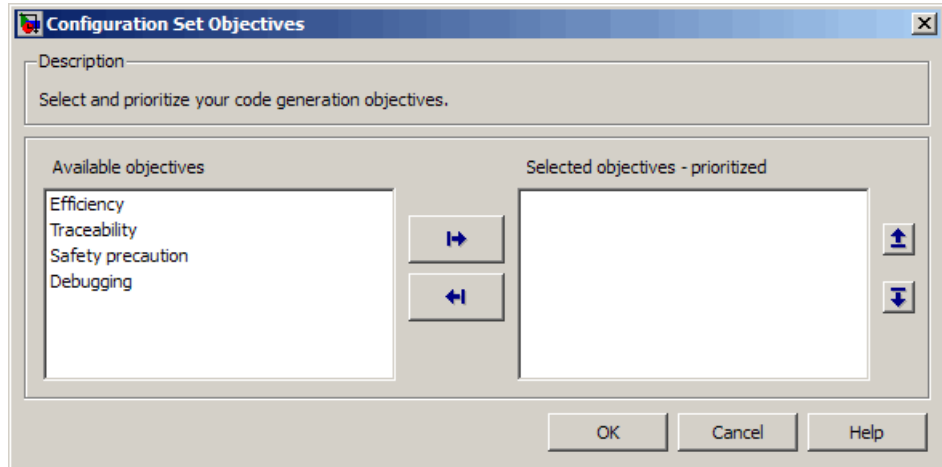
This button appears only for ERT-based targets.

See Also

- “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.
- “Configuring Code Generation Objectives” in the Real-Time Workshop User’s Guide.

Configuration Set Objectives Dialog Box

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



Settings

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

Objectives. List of available objectives.

Efficiency — Configure code generation settings to reduce RAM, ROM, and execution time.

Traceability — Configure code generation settings to provide mapping between model elements and code.

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

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

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

Dependency

This dialog box appears only for ERT-based targets.

Command-Line Information

Parameter: 'ObjectivePriorities'

Type: cell array of strings; any combination of the available values

Value: {' '} | {'Efficiency'} | {'Traceability'} | {'Safety precaution'} | {'Debugging'}

Default: {' '}

See Also

- “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.
- “Configuring Code Generation Objectives” in the Real-Time Workshop User’s Guide.

Check model

Runs the Code Generation Advisor checks.

Settings

- 1 Specify code generation objectives using the **Select objective** parameter (available with GRT-based targets) or in the Configuration Set Objectives dialog box, by clicking **Set objectives** (available with ERT-based targets).
- 2 Click **Check model**. The Code Generation Advisor runs the code generation objectives checks.

Dependency

You must specify objectives before checking the model.

See Also

- “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.
- “Configuring Code Generation Objectives” in the Real-Time Workshop User’s Guide.

Check model before generating code

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

Settings

Default: Off

Off

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

On (proceed with warnings)

Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the Real-Time Workshop software continues generating code. The code generation report summary and file headers indicate the specified objectives and the validation result.

On (stop for warnings)

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

Command-Line Information

Parameter: CheckMdlBeforeBuild

Type: string

Value: 'Off' | 'Warning' | 'Error'

Default: 'Off'

Recommended Settings

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

See Also

- “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.
- “Configuring Code Generation Objectives” in the Real-Time Workshop User’s Guide.

Generate code only

Specify code generation versus an executable build.

Settings

Default: off



On

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



Off

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

Tip

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

Dependency

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

Command-Line Information

Parameter: GenCodeOnly

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | Off |
| Traceability | No impact |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

Customizing Post Code Generation Build Processing

Build/Generate code

Start the build or code generation process.

Tip

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

Dependency

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

Command-Line Information

Command: `rtwbuild`

Type: `string`

Value: `'modelName'`

Recommended Settings

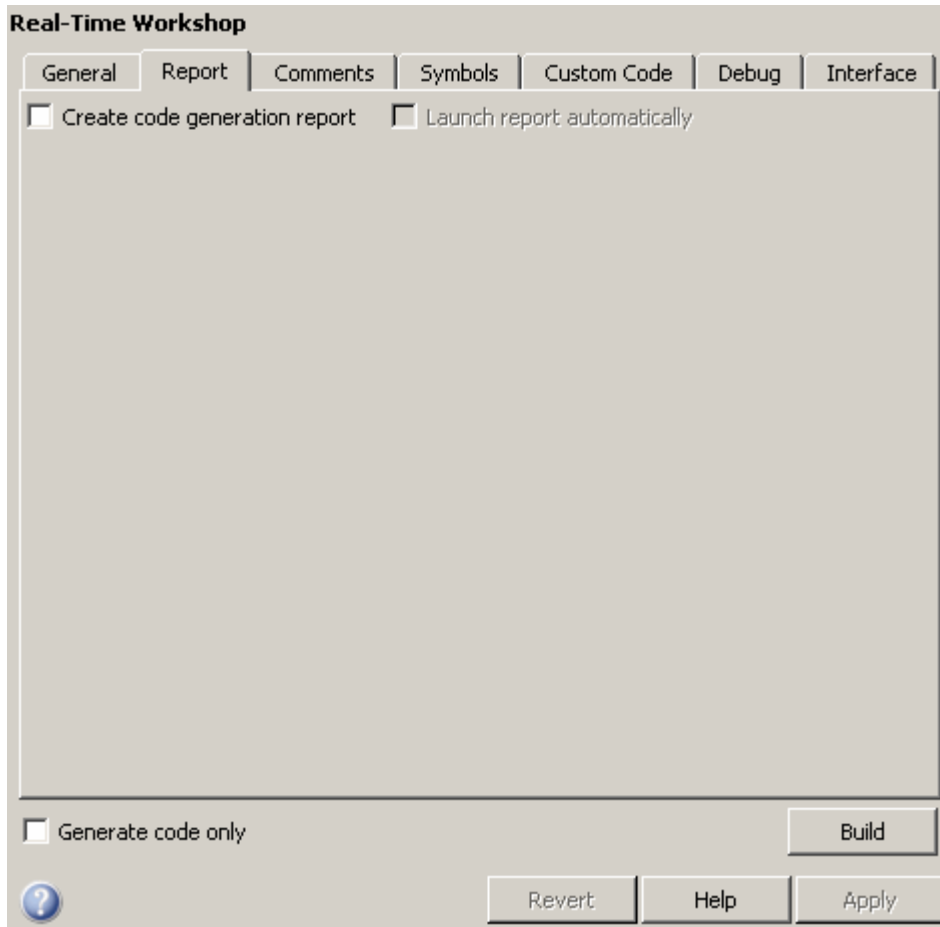
| Application | Setting |
|-------------------|--------------|
| Debugging | Build |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

Initiating the Build Process

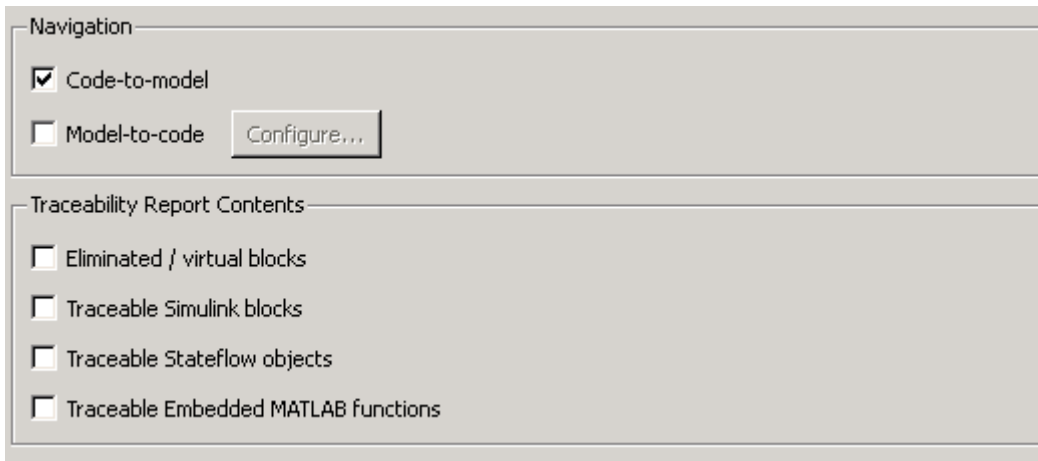
Real-Time Workshop Pane: Report

The Real-Time Workshop Report pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select a GRT-based target.



The Real-Time Workshop Report pane includes the following parameters when the Real-Time Workshop product is installed on your system and

you select an ERT-based target. ERT-based target parameters require a Real-Time Workshop Embedded Coder license when generating code.



The screenshot shows a configuration dialog box with two main sections:

- Navigation:**
 - Code-to-model
 - Model-to-code [Configure...](#)
- Traceability Report Contents:**
 - Eliminated / virtual blocks
 - Traceable Simulink blocks
 - Traceable Stateflow objects
 - Traceable Embedded MATLAB functions

In this section...

“Real-Time Workshop: Report Tab Overview” on page 6-39

“Create code generation report” on page 6-40

“Launch report automatically” on page 6-43

“Code-to-model” on page 6-45

“Model-to-code” on page 6-47

“Configure” on page 6-49

“Eliminated / virtual blocks” on page 6-50

“Traceable Simulink blocks” on page 6-52

“Traceable Stateflow objects” on page 6-54

“Traceable Embedded MATLAB functions” on page 6-56

Real-Time Workshop: Report Tab Overview

Control the Code Generation report that the Real-Time Workshop software automatically creates.

Configuration

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

See Also

- Generate HTML Report

If you have a Real-Time Workshop Embedded Coder license, see also [Creating and Using a Code Generation Report](#).

- “Real-Time Workshop Pane: Report” on page 6-37

Create code generation report

Document generated code in an HTML report.

Settings

Default: Off



On

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

- The **Summary** section lists version and date information. The **Configuration Settings at the Time of Code Generation** link opens a noneditable view of the Configuration Parameters dialog that shows the Simulink model settings, including TLC options, at the time of code generation.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data (requires a Real-Time Workshop Embedded Coder license and the ERT target).
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts**, providing a complete mapping between model elements and code (requires a Real-Time Workshop Embedded Coder license and the ERT target).

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

- Global variable instances are hyperlinked to their definitions.
- If you selected the traceability option **Code-to-model**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view

the relevant blocks or subsystems in a Simulink model window (requires a Real-Time Workshop Embedded Coder license and the ERT target).

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



Off

Does not generate a summary of files.

Dependency

This parameter enables and selects

- **Launch report automatically**
- **Code-to-model** (ERT target)

This parameter enables

- **Model-to-code** (ERT target)
- **Eliminated / virtual blocks** (ERT target)
- **Traceable Simulink blocks** (ERT target)
- **Traceable Stateflow objects** (ERT target)
- **Traceable Embedded MATLAB functions** (ERT target)

Command-Line Information

Parameter: GenerateReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

Generate HTML Report

If you have a Real-Time Workshop Embedded Coder license, see also [Creating and Using a Code Generation Report](#).

Launch report automatically

Specify whether to display Code Generation reports automatically.

Settings

Default: Off



On

Displays the Code Generation report automatically in a new browser window.



Off

Does not display the Code Generation report, but the report is still available in the html directory.

Dependency

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

Command-Line Information

Parameter: LaunchReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

Generate HTML Report

If you have a Real-Time Workshop Embedded Coder license, see also [Creating and Using a Code Generation Report](#).

Code-to-model

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

Settings

Default: Off



On

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



Off

Omits hyperlinks from the generated report.

Tip

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled and selected by **Create code generation report**.
- You must select **Include comments** on the **Real-Time Workshop > Comments** tab to use this parameter.

Command-Line Information

Parameter: IncludeHyperlinkInReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

Creating and Using a Code Generation Report.

Model-to-code

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

Settings

Default: Off



On

Includes model-to-code highlighting support in the Code Generation report. To highlight the generated code for a Simulink block, Stateflow object, or Embedded MATLAB script in the Code Generation report, right-click the item and select **Real-Time Workshop > Navigate to Code**.



Off

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

Tip

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.
- This parameter selects:
 - **Eliminated / virtual blocks**
 - **Traceable Simulink blocks**
 - **Traceable Stateflow objects**

- **Traceable Embedded MATLAB functions**
- You must select the following parameters to use this parameter:
 - **Include comments** on the **Real-Time Workshop > Comments** tab
 - At least one of the following:
 - **Eliminated / virtual blocks**
 - **Traceable Simulink blocks**
 - **Traceable Stateflow objects**
 - **Traceable Embedded MATLAB functions**

Command-Line Information

Parameter: GenerateTraceInfo

Type: Boolean

Value: on | off

Default: off

Recommended Settings

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

See Also

Creating and Using a Code Generation Report.

Configure

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

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled by **Model-to-code**.

See Also

Creating and Using a Code Generation Report.

Eliminated / virtual blocks

Include summary of eliminated and virtual blocks in Code Generation report.

Settings

Default: Off

- On
Includes a summary of eliminated and virtual blocks in the Code Generation report.
- Off
Does not include a summary of eliminated and virtual blocks.

Dependencies

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

Command-Line Information

Parameter: GenerateTraceReport

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|---------|
| Debugging | On |
| Traceability | On |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | On |

See Also

Creating and Using a Code Generation Report.

Traceable Simulink blocks

Include summary of Simulink blocks in Code Generation report.

Settings

Default: Off

- On
Includes a summary of Simulink blocks and the corresponding code location in the Code Generation report.
- Off
Does not include a summary of Simulink blocks.

Dependencies

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

Command-Line Information

Parameter: GenerateTraceReportSl

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|---------|
| Debugging | On |
| Traceability | On |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | On |

See Also

Creating and Using a Code Generation Report.

Traceable Stateflow objects

Include summary of Stateflow objects in Code Generation report.

Settings

Default: Off



On

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



Off

Does not include a summary of Stateflow objects.

Dependencies

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

Command-Line Information

Parameter: GenerateTraceReportSf

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|---------|
| Debugging | On |
| Traceability | On |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | On |

See Also

Creating and Using a Code Generation Report.

Traceability of Stateflow Objects in Generated Code.

Traceable Embedded MATLAB functions

Include summary of Embedded MATLAB functions in Code Generation report.

Settings

Default: Off



On

Includes a summary of Embedded MATLAB functions and corresponding code locations in the Code Generation report.



Off

Does not include a summary of Embedded MATLAB functions.

Dependencies

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

Command-Line Information

Parameter: GenerateTraceReportEm1

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|---------|
| Debugging | On |
| Traceability | On |

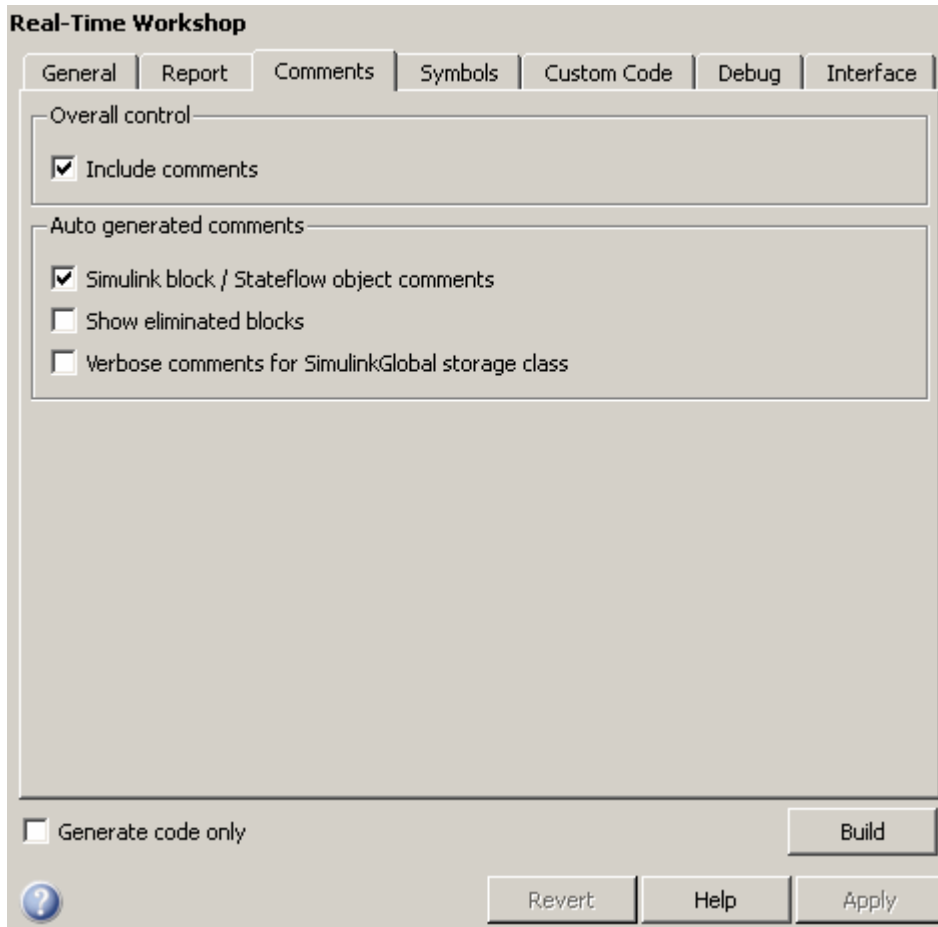
| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | On |

See Also

Creating and Using a Code Generation Report.

Real-Time Workshop Pane: Comments

The Real-Time Workshop Comments pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select a GRT-based target.



The Real-Time Workshop Comments pane includes the following parameters when the Real-Time Workshop product is installed on your system and

you select an ERT-based target. ERT-based target parameters require a Real-Time Workshop Embedded Coder license when generating code.

Custom comments

| | |
|---|---|
| <input type="checkbox"/> Simulink block descriptions | <input type="checkbox"/> Stateflow object descriptions |
| <input type="checkbox"/> Simulink data object descriptions | <input type="checkbox"/> Requirements in block comments |
| <input type="checkbox"/> Custom comments (MPT objects only) | |

In this section...

“Real-Time Workshop: Comments Tab Overview” on page 6-60

“Include comments” on page 6-61

“Simulink block / Stateflow object comments” on page 6-62

“Show eliminated blocks” on page 6-63

“Verbose comments for SimulinkGlobal storage class” on page 6-64

“Simulink block descriptions” on page 6-65

“Simulink data object descriptions” on page 6-67

“Custom comments (MPT objects only)” on page 6-69

“Custom comments function” on page 6-71

“Stateflow object descriptions” on page 6-73

“Requirements in block comments” on page 6-75

Real-Time Workshop: Comments Tab Overview

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

See Also

“Real-Time Workshop Pane: Comments” on page 6-58

Include comments

Specify which comments are in generated files.

Settings

Default: on



On

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



Off

Omits comments from the generated files.

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: GenerateComments

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

Simulink block / Stateflow object comments

Specify whether to insert Simulink block and Stateflow object comments.

Settings

Default: on



On

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



Off

Suppresses comments.

Dependency

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

Command-Line Information

Parameter: SimulinkBlockComments

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

Show eliminated blocks

Specify whether to insert eliminated block's comments.

Settings

Default: off



On

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



Off

Suppresses statements.

Dependency

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

Command-Line Information

Parameter: ShowEliminatedStatement

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Verbose comments for SimulinkGlobal storage class

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

Settings

Default: off



On

Generates parameter comments regardless of the number of parameters.



Off

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

Dependency

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

Command-Line Information

Parameter: ForceParamTrailComments

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Simulink block descriptions

Specify whether to insert descriptions of blocks into generated code as comments.

Settings

Default: off



On

Includes the following comments in the generated code for each block in the model, with the exception of virtual blocks and blocks removed due to block reduction:

- The block name at the start of the code, regardless of whether you select **Simulink block / Stateflow object comments**
- Text specified in the **Description** field of each Block Parameter dialog box

The block names and descriptions can include international (non-US-ASCII) characters.



Off

Suppresses the generation of block name and description comments in the generated code.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: InsertBlockDesc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

Support for International (Non-US-ASCII) Characters

Simulink data object descriptions

Specify whether to insert descriptions of Simulink data objects into generated code as comments.

Settings

Default: off



On

Inserts contents of the **Description** field in the Model Explorer Object Properties pane for each Simulink data object (signal, parameter, and bus objects) in the generated code as comments.

The descriptions can include international (non-US-ASCII) characters.



Off

Suppresses the generation of data object property descriptions as comments in the generated code.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: SimulinkDataObjDesc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|---------|
| Debugging | On |
| Traceability | On |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | No impact |

Custom comments (MPT objects only)

Specify whether to include custom comments for module packaging tool (MPT) signal and parameter data objects in generated code.

Settings

Default: off



On

Inserts comments just above the identifiers for signal and parameter MPT objects in generated code.



Off

Suppresses the generation of custom comments for signal and parameter identifiers.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter requires that you include the comments in a function defined in an M-file or TLC file that you specify with **Custom comments function**.

Command-Line Information

Parameter: EnableCustomComments

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|-------------|---------|
| Debugging | On |

| Application | Setting |
|--------------------|----------------|
| Traceability | On |
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

Adding Custom Comments

Custom comments function

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

Settings

Default: ''

Enter the name of the M-file or TLC file for the function that includes the comments to be inserted of your MPT signal and parameter objects. You can specify the file name directly or click **Browse** and search for a file.

Tip

You might use this option to insert comments that document some or all of an object's property values.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled by **Custom comments (MPT objects only)**.

Command-Line Information

Parameter: CustomCommentsFcn

Type: string

Value: any valid file name

Default: ''

Recommended Settings

| Application | Setting |
|--------------|---------------------|
| Debugging | Any valid file name |
| Traceability | Any valid file name |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

Adding Custom Comments

Stateflow object descriptions

Specify whether to insert descriptions of Stateflow objects into generated code as comments.

Settings

Default: off



On

Inserts descriptions of Stateflow states, charts, transitions, and graphical functions into generated code as comments. The descriptions come from the **Description** field in Object Properties pane in the Model Explorer for these Stateflow objects. The comments appear just above the code generated for each object.

The descriptions can include international (non-US-ASCII) characters.



Off

Suppresses the generation of comments for Stateflow objects.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Stateflow license.

Command-Line Information

Parameter: SFDataObjDesc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|---------|
| Debugging | On |
| Traceability | On |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

Support for International (Non-US-ASCII) Characters

Requirements in block comments

Specify whether to include requirement descriptions assigned to Simulink blocks in generated code as comments.

Settings

Default: off



On

Inserts the requirement descriptions that you assign to Simulink blocks into the generated code as comments. The Real-Time Workshop software includes the requirement descriptions in the generated code in the following locations.

| Model Element | Requirement Description Location |
|-----------------------|--|
| Model | In the main header file <i>model.h</i> |
| Nonvirtual subsystems | At the call site for the subsystem |
| Virtual subsystems | At the call site of the closest nonvirtual parent subsystem. If a virtual subsystem has no nonvirtual parent, requirement descriptions are located in the main header file for the model, <i>model.h</i> . |
| Nonsubsystem blocks | In the generated code for the block |

The requirement text can include international (non-US-ASCII) characters.



Off

Suppresses the generation of comments for block requirement descriptions.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires Real-Time Workshop Embedded Coder and Simulink® Verification and Validation™ licenses when generating code.

Command-Line Information

Parameter: ReqsInCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

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

Real-Time Workshop Pane: Symbols

The Real-Time Workshop Symbols pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select a GRT-based target.

The image shows a software configuration window titled "Real-Time Workshop". At the top, there are several tabs: "General", "Report", "Comments", "Symbols" (which is selected), "Custom Code", "Debug", and "Interface". Below the tabs, the "Auto-generated identifier naming rules" section contains a text input field labeled "Maximum identifier length:" with the value "31". The "Reserved names" section has a checkbox labeled "Use the same reserved names as Simulation Target" which is unchecked. Below this checkbox is a text area labeled "Reserved names:" which is currently empty. At the bottom left, there is a checkbox labeled "Generate code only" which is also unchecked. At the bottom right, there are three buttons: "Build", "Revert", and "Apply". A help icon (a question mark in a circle) is located in the bottom left corner of the window.

The Real-Time Workshop Symbols pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select an ERT-based target. ERT-based target parameters require a Real-Time Workshop Embedded Coder license when generating code.

Auto-generated identifier naming rules

Identifier format control

| | |
|-------------------------------|-------------|
| Global variables: | rt\$N\$M |
| Global types: | \$N\$M |
| Field name of global types: | \$N\$M |
| Subsystem methods: | \$N\$M\$F |
| Subsystem method arguments: | rt\$I\$N\$M |
| Local temporary variables: | \$N\$M |
| Local block output variables: | rtb_ \$N\$M |
| Constant macros: | \$N\$M |

Minimum mangle length: 1

Maximum identifier length: 31

Generate scalar inlined parameters as: Literals

Simulink data object naming rules

| | |
|-------------------|------|
| Signal naming: | None |
| Parameter naming: | None |
| #define naming: | None |

Reserved names

Use the same reserved names as Simulation Target

Reserved names:

In this section...

“Real-Time Workshop: Symbols Tab Overview” on page 6-81

“Global variables” on page 6-82

“Global types” on page 6-84

“Field name of global types” on page 6-87

“Subsystem methods” on page 6-89

“Subsystem method arguments” on page 6-92

“Local temporary variables” on page 6-94

“Local block output variables” on page 6-96

“Constant macros” on page 6-98

“Minimum mangle length” on page 6-100

“Maximum identifier length” on page 6-102

“Generate scalar inlined parameter as” on page 6-104

“Signal naming” on page 6-105

“M-function” on page 6-107

“Parameter naming” on page 6-109

“#define naming” on page 6-111

“Use the same reserved names as Simulation Target” on page 6-113

“Reserved names” on page 6-114

Real-Time Workshop: Symbols Tab Overview

Select the automatically generated identifier naming rules.

See Also

- Symbols Options
- “Real-Time Workshop Pane: Symbols” on page 6-77

Global variables

Customize generated global variable identifiers.

Settings

Default: \$R\$N\$M

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

| Token | Description |
|-------|--|
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. |
| \$R | Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model,

the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrGlobalVar

Type: string

Value: any valid combination of tokens

Default: '\$R\$N\$M'

Recommended Settings

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

See Also

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

Global types

Customize generated global type identifiers.

Settings

Default: \$N\$R\$M

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

| Token | Description |
|-------|--|
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. |
| \$R | Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model,

the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

- Name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify `$R`, the code generator includes the model name in the `typedef`.
- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: `CustomSymbolStrType`

Type: `string`

Value: any valid combination of tokens

Default: `'NR$M'`

Recommended Settings

| Application | Setting |
|-------------------|---------------------------------|
| Debugging | No impact |
| Traceability | Any valid combination of tokens |
| Efficiency | No impact |
| Safety precaution | <code>\$N\$R\$M</code> |

See Also

- Specifying Identifier Formats

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

Field name of global types

Customize generated field names of global types.

Settings

Default: \$N\$M

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

| Token | Description |
|-------|---|
| \$A | Insert data type acronym (for example, i32 for long integers) into signal and work vector identifiers. |
| \$H | Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string root_. For blocks at the subsystem level, the tag is of the form sN_, where N is a unique system number assigned by the Simulink software. |
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- The **Maximum identifier length** setting does not apply to type definitions.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrField

Type: string

Value: any valid combination of tokens

Default: '\$N\$M'

Recommended Settings

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

See Also

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

Subsystem methods

Customize generated global type identifiers.

Settings

Default: \$R\$N\$M\$F

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

| Token | Description |
|-------|---|
| \$F | Insert method name (for example, _Update for update method). Empty for Stateflow functions. |
| \$H | Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string root_. For blocks at the subsystem level, the tag is of the form sN_, where N is a unique system number assigned by the Simulink software. Empty for Stateflow functions. |
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. |
| \$R | Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model, the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.
- Name mangling conventions do not apply to type names (that is, typedef statements) generated for global data types. The **Maximum identifier length** setting does not apply to type definitions. If you specify \$R, the code generator includes the model name in the typedef.
- This option does not affect objects (such as signals and parameters) that have a storage class other than Auto (such as ImportedExtern or ExportedGlobal).

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrFcn

Type: string

Value: any valid combination of tokens

Default: '\$R\$N\$M\$F'

Recommended Settings

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

See Also

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

Subsystem method arguments

Customize generated function argument names for reusable subsystems.

Settings

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

| Token | Description |
|-------|---|
| \$I | Insert an u if the argument is an input. Insert a y if the argument is an output. Optional. |
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. Recommended to ensure readability of generated code. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.

Dependencies

This parameter:

- Appears only for ERT-based targets.

- Requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrFcnArg

Type: string

Value: any valid combination of tokens

Default: 'rtu_\$\$M' or 'rty_\$\$M'

Recommended Settings

| Application | Setting |
|-------------------|----------------------------------|
| Debugging | No impact |
| Traceability | Any valid combinations of tokens |
| Efficiency | No impact |
| Safety precaution | rtu_\$\$M or rty_\$\$M |

See Also

- “Real-Time Workshop Pane: Symbols” on page 6-77
- Specifying Identifier Formats
- Name Mangling
- Identifier Format Control Parameters Limitations

Local temporary variables

Customize generated local temporary variable identifiers.

Settings

Default: \$N\$M

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

| Token | Description |
|-------|--|
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. |
| \$R | Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Required for model referencing. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model,

the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrTmpVar

Type: string

Value: any valid combination of tokens

Default: '\$N\$M'

Recommended Settings

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

See Also

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

Local block output variables

Customize generated local block output variable identifiers.

Settings

Default: `rtb_$$M`

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

| Token | Description |
|-------|---|
| \$A | Insert data type acronym (for example, <code>i32</code> for long integers) into signal and work vector identifiers. |
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, `Gain1`, `Gain2`...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- This option does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrBlkIO

Type: string

Value: any valid combination of tokens

Default: 'rtb_\$\$M'

Recommended Settings

| Application | Setting |
|-------------------|---------------------------------|
| Debugging | No impact |
| Traceability | Any valid combination of tokens |
| Efficiency | No impact |
| Safety precaution | rtb_\$\$M |

See Also

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

Constant macros

Customize generated constant macro identifiers.

Settings

Default: \$R\$N\$M

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

| Token | Description |
|-------|--|
| \$M | Insert name mangling string if required to avoid naming collisions. Required. |
| \$N | Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated. |
| \$R | Insert root model name into identifier, replacing any unsupported characters with the underscore () character. Required for model referencing. |

Tips

- Avoid name collisions in general. One way is to avoid using default block names (for example, Gain1, Gain2...) when your model has many blocks of the same type.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the identifiers you expect to generate. Reserve at least three characters for a name mangling string.
- If you specify \$R, the value you specify for **Maximum identifier length** must be large enough to accommodate full expansions of the \$R and \$M tokens.
- When a name conflict occurs between an identifier within the scope of a higher-level model and an identifier within the scope of a referenced model,

the code generator preserves the identifier from the referenced model. Name mangling is performed on the identifier in the higher-level model.

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

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: CustomSymbolStrMacro

Type: string

Value: any valid combination of tokens

Default: '\$R\$N\$M'

Recommended Settings

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

See Also

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

Minimum mangle length

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

Settings

Default: 1

Specify an integer value that indicates the minimum number of characters the code generator is to use when generating a name mangling string. As necessary, the minimum value automatically increases during code generation as a function of the number of collisions. A larger value reduces the chance of identifier disturbance when you modify the model.

Tips

- Minimize disturbance to the generated code during development, by specifying a value of 4. This value is conservative and safe; it allows for over 1.5 million collisions for a particular identifier before the mangle length increases.
- Set the value to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

Dependency

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: MangleLength

Type: integer

Value: any valid value

Default: 1

Recommended Settings

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

See Also

- Name Mangling
- Traceability
- Minimizing Name Mangling

Maximum identifier length

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

Settings

Default: 31

Minimum: 31

Maximum: 256

You can use this parameter to limit the number of characters in function, type definition, and variable names.

Tips

- Consider increasing identifier length for models having a deep hierarchical structure.
- When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate the root model name and the name mangling string (if any). A code generation error occurs if **Maximum identifier length** is too small.
- This parameter must be the same for both top-level and referenced models.
- When a name conflict occurs between a symbol within the scope of a higher level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. Name mangling is performed on the symbol from the higher level model.

Command-Line Information

Parameter: MaxIdLength

Type: integer

Value: any valid value

Default: 31

Recommended Settings

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

See Also

Generating Code for Model Referencing

Generate scalar inlined parameter as

Control expression of scalar inlined parameter values in the generated code.

Settings

Default: Literals

Literals

Generates scalar inlined parameters as numeric constants. This setting can help with debugging TLC code, as it makes it easy to search for parameter values in the generated code.

Macros

Generates scalar inlined parameters as variables with `#define` macros. This setting makes generated code more readable.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: `InlinedPrmAccess`

Type: string

Value: 'Literals' | 'Macros'

Default: 'Literals'

Recommended Settings

| Application | Setting |
|-------------------|-----------|
| Debugging | No impact |
| Traceability | Macros |
| Efficiency | Literals |
| Safety precaution | No impact |

Signal naming

Specify rules for naming signals in generated code.

Settings

Default: None

None

Makes no change to signal names when creating corresponding identifiers in generated code. Signal identifiers in the generated code match the signal names that appear in the model.

Force upper case

Uses all uppercase characters when creating identifiers for signal names in the generated code.

Force lower case

Uses all lowercase characters when creating identifiers for signal names in the generated code.

Custom M-function

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Setting this parameter to Custom M-function enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Limitation

This parameter has no effect on signal names that are specified by an embedded signal object created using the **Real Time Workshop** tab of a signal's **Signal Properties** dialog box. See “Applying a CSC Using an Embedded Signal Object” for information about embedded signal objects.

Command-Line Information

Parameter: SignalNamingRule

Type: string

Value: 'None' | 'UpperCase' | 'LowerCase' | 'Custom'

Default: 'None'

Recommended Settings

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

See Also

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

M-function

Specify rule for naming identifiers in generated code.

Settings

Default: ''

Enter the name of an M-file that contains the naming rule to be applied to signal, parameter, or `#define` parameter identifiers in generated code. Examples of rules you might program in such an M-file function include:

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

Tip

M-file must be in the MATLAB path.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled by **Signal naming**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: DefineNamingFcn

Type: string

Value: any M-file

Default: ''

Recommended Settings

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

See Also

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

Parameter naming

Specify rule for naming parameters in generated code.

Settings

Default: None

None

Makes no change to parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case

Uses all uppercase characters when creating identifiers for parameter names in the generated code.

Force lower case

Uses all lowercase characters when creating identifiers for parameter names in the generated code.

Custom M-function

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Setting this parameter to Custom M-function enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: ParamNamingRule

Type: string

Value: 'None' | 'UpperCase' | 'LowerCase' | 'Custom'

Default: 'None'

Recommended Settings

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

See Also

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

#define naming

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

Settings

Default: None

None

Makes no change to `#define` parameter names when creating corresponding identifiers in generated code. Parameter identifiers in the generated code match the parameter names that appear in the model.

Force upper case

Uses all uppercase characters when creating identifiers for `#define` parameter names in the generated code.

Force lower case

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

Custom M-function

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

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Setting this parameter to `Custom M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `DefineNamingRule`

Type: `string`

Value: `'None' | 'UpperCase' | 'LowerCase' | 'Custom'`

Default: `'None'`

Recommended Settings

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

See Also

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

Use the same reserved names as Simulation Target

Specify whether to use the same reserved names as those specified in the **Simulation Target > Symbols** pane.

Settings

Default: Off



On

Enables using the same reserved names as those specified in the **Simulation Target > Symbols** pane.



Off

Disables using the same reserved names as those specified in the **Simulation Target > Symbols** pane.

Command-Line Information

Parameter: UseSimReservedNames

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Reserved names

Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

Settings

Default: {}

This action changes the names of variables or functions in the generated code to avoid name conflicts with identifiers in custom code. Reserved names must be shorter than 256 characters.

Tips

- Do not enter Real-Time Workshop keywords since these names cannot be changed in the generated code. For a list of keywords to avoid, see “Reserved Keywords” in the *Real-Time Workshop User’s Guide*.
- Start each reserved name with a letter or an underscore to prevent error messages.
- Each reserved name must contain only letters, numbers, or underscores.
- Separate the reserved names using commas or spaces.
- You can also specify reserved names by using the command line:

```
config_param_object.set_param('ReservedNameArray',  
{'abc', 'xyz'})
```

where *config_param_object* is the object handle to the model settings in the Configuration Parameters dialog box.

Command-Line Information

Parameter: ReservedNameArray

Type: string array

Value: any reserved names shorter than 256 characters

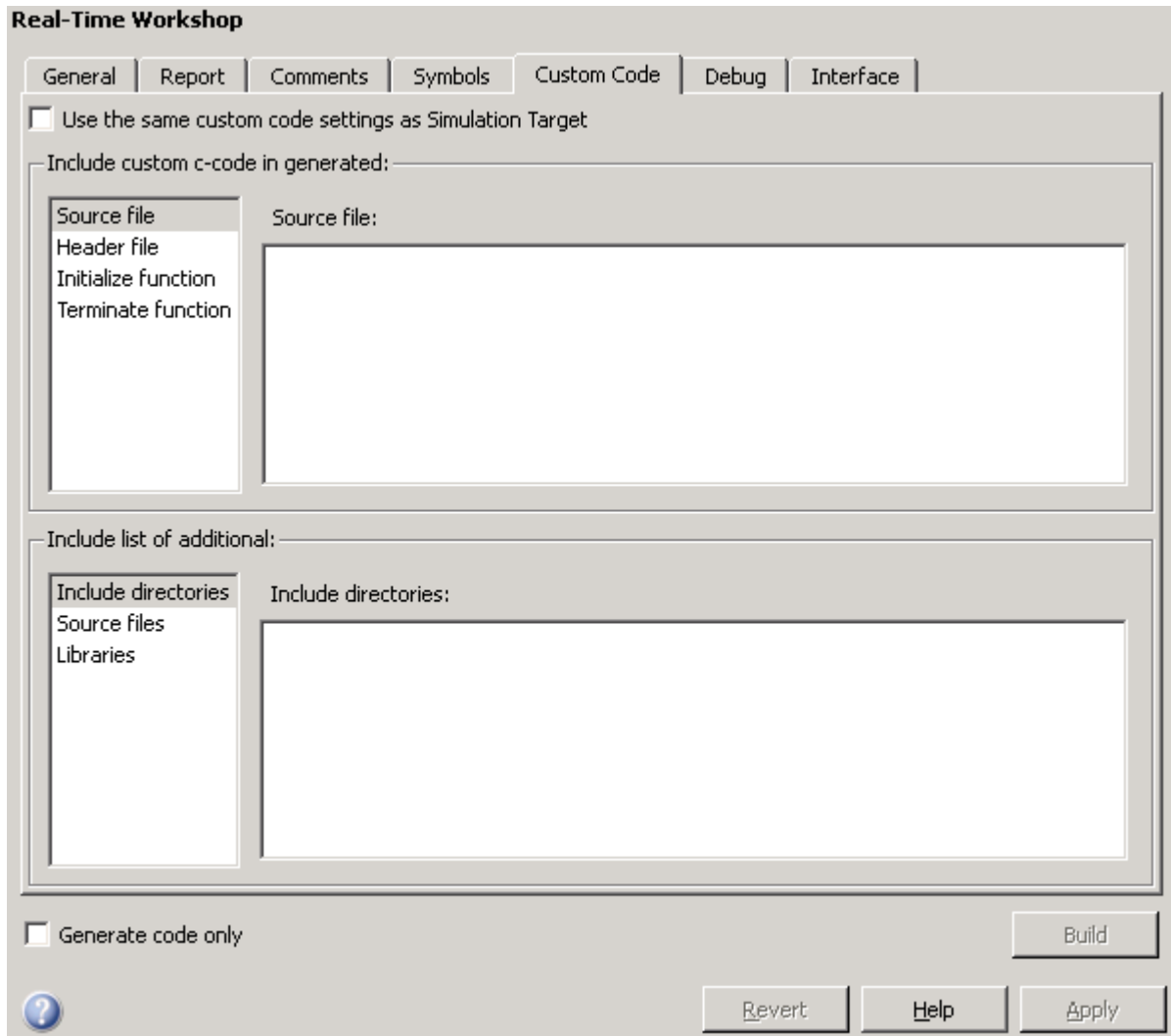
Default: {}

Recommended Settings

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

Real-Time Workshop Pane: Custom Code

The Real-Time Workshop Custom Code pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select a GRT- or ERT-based target.



In this section...

“Real-Time Workshop: Custom Code Tab Overview” on page 6-118

“Use the same custom code settings as Simulation Target” on page 6-119

“Use local custom code settings (do not inherit from main model)” on page 6-120

“Source file” on page 6-122

“Header file” on page 6-123

“Initialize function” on page 6-124

“Terminate function” on page 6-125

“Include directories” on page 6-126

“Source files” on page 6-128

“Libraries” on page 6-130

Real-Time Workshop: Custom Code Tab Overview

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

Configuration

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

See Also

- Configuring Custom Code
- “Real-Time Workshop Pane: Custom Code” on page 6-116

Use the same custom code settings as Simulation Target

Specify whether to use the same custom code settings as those in the **Simulation Target > Custom Code** pane.

Settings

Default: Off



On

Enables using the same custom code settings as those in the **Simulation Target > Custom Code** pane.



Off

Disables using the same custom code settings as those in the **Simulation Target > Custom Code** pane.

Command-Line Information

Parameter: RTWUseSimCustomCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Use local custom code settings (do not inherit from main model)

Specify if a library model can use custom code settings that are unique from the main model.

Settings

Default: Off

- On
Enables a library model to use custom code settings that are unique from the main model.
- Off
Disables a library model from using custom code settings that are unique from the main model.

Dependency

This parameter is available only for library models that contain Embedded MATLAB Function blocks, Stateflow charts, or Truth Table blocks. To access this parameter, select **Tools > Open RTW Target** in the Embedded MATLAB Editor or Stateflow Editor for your library model.

Command-Line Information

Parameter: RTWUseLocalCustomCode
Type: string
Value: 'on' | 'off'
Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | No impact |
| Traceability | No impact |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | No impact |

Source file

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

Settings

Default: ''

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

Command-Line Information

Parameter: CustomSourceCode

Type: string

Value: any source file name

Default: ''

Recommended Settings

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

Header file

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

Settings

Default: ''

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

Command-Line Information

Parameter: CustomHeaderCode

Type: string

Value: any header file name

Default: ''

Recommended Settings

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

Initialize function

Specify code appearing in an initialize function.

Settings

Default: ''

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

Command-Line Information

Parameter: CustomInitializer

Type: string

Value: any code

Default: ''

Recommended Settings

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

Terminate function

Specify code appearing in a terminate function.

Settings

Default: ''

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

Dependency

A terminate function is generated only if you select the **Terminate function required** check box on the **Real-Time Workshop** pane, **Interface** tab.

Command-Line Information

Parameter: CustomTerminator

Type: string

Value: any code

Default: ''

Recommended Settings

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

Include directories

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

Settings

Default: ' '

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

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

Note If you specify a Windows path string containing one or more spaces, you must enclose the string in double quotes. For example, the second and third path strings in the **Include directories** entry below must be double-quoted:

```
C:\Project "C:\Custom Files" "C:\Library Files"
```

If you set the equivalent command-line parameter `CustomInclude`, each path string containing spaces must be separately double-quoted within the single-quoted third argument string, for example,

```
>> set_param('mymodel', 'CustomInclude', ...  
            'C:\Project "C:\Custom Files" "C:\Library Files"')
```

Command-Line Information

Parameter: `CustomInclude`

Type: string

Value: any directory file name

Default: ' '

Recommended Settings

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

Source files

Specify a list of source files to compile and link with the generated code.

Settings

Default: ''

Enter a space-separated list of source files to compile and link with the generated code.

Note If you specify a Windows file name string containing one or more spaces, you must enclose the string in double quotes. For example, the second and third name strings in the **Source files** entry below must be double-quoted:

```
algs.c "div alg.c" "sin alg.c"
```

If you set the equivalent command-line parameter `CustomSource`, each file name string containing spaces must be separately double-quoted within the single-quoted third argument string, for example,

```
>> set_param('mymodel', 'CustomSource', ...  
            'algs.c "div alg.c" "sin alg.c"')
```

Tip

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

Command-Line Information

Parameter: `CustomSource`

Type: string

Value: any source file name

Default: ''

Recommended Settings

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

Libraries

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

Settings

Default: ''

Enter a space-separated list of static library files to link with the generated code.

Note If you specify a Windows file name string containing one or more spaces, you must enclose the string in double quotes. For example, the second and third name strings in the **Libraries** entry below must be double-quoted:

```
mathutil.lib "op utils.lib" "fx utils.lib"
```

If you set the equivalent command-line parameter `CustomLibrary`, each file name string containing spaces must be separately double-quoted within the single-quoted third argument string, for example,

```
>> set_param('mymodel', 'CustomLibrary', ...  
            'mathutil.lib "op utils.lib" "fx utils.lib"')
```

Tip

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

Command-Line Information

Parameter: `CustomLibrary`

Type: string

Value: any library file name

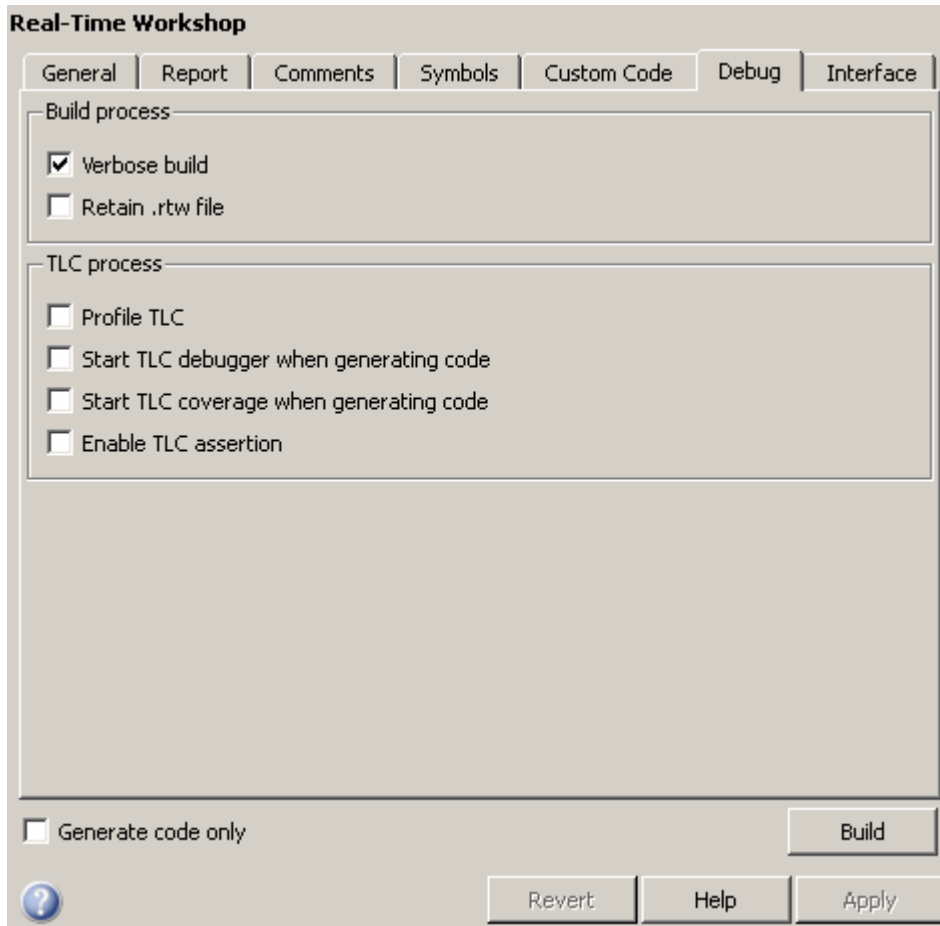
Default: ''

Recommended Settings

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

Real-Time Workshop Pane: Debug

The Real-Time Workshop Debug pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select a GRT- or ERT-based target.



In this section...

“Real-Time Workshop: Debug Tab Overview” on page 6-134

“Verbose build” on page 6-135

“Retain .rtw file” on page 6-136

“Profile TLC” on page 6-137

“Start TLC debugger when generating code” on page 6-138

“Start TLC coverage when generating code” on page 6-139

“Enable TLC assertion” on page 6-140

Real-Time Workshop: Debug Tab Overview

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

See Also

- Troubleshooting the Build Process
- “Real-Time Workshop Pane: Debug” on page 6-132

Verbose build

Display code generation progress.

Settings

Default: on



On

The MATLAB Command Window displays progress information indicating code generation stages and compiler output during code generation.



Off

Does not display progress information.

Command-Line Information

Parameter: RTWVerbose

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

Retain .rtw file

Specify *model*.rtw file retention.

Settings

Default: off



On

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



Off

Deletes the *model*.rtw from the build directory at the end of the build process.

Command-Line Information

Parameter: RetainRTWFile

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Profile TLC

Profile the execution time of TLC files.

Settings

Default: off



On

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



Off

Does not profile the performance.

Command-Line Information

Parameter: ProfileTLC

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Start TLC debugger when generating code

Specify use of the TLC debugger

Settings

Default: off

- On
The TLC debugger starts during code generation.
- Off
Does not start the TLC debugger.

Tips

- You can also start the TLC debugger by entering the `-dc` argument into the **System target file** field.
- To invoke the debugger and run a debugger script, enter the `-df filename` argument into the **System target file** field.

Command-Line Information

Parameter: TLCDebug

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Start TLC coverage when generating code

Generate the TLC execution report.

Settings

Default: off



On

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



Off

Does not generate a report.

Tip

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

Command-Line Information

Parameter: TLCCoverage

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Enable TLC assertion

Produce the TLC stack trace

Settings

Default: off



On

The build process halts if any user-supplied TLC file contains an %assert directive that evaluates to FALSE.



Off

The build process ignores TLC assertion code.

Command-Line Information

Parameter: TLCAssert

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

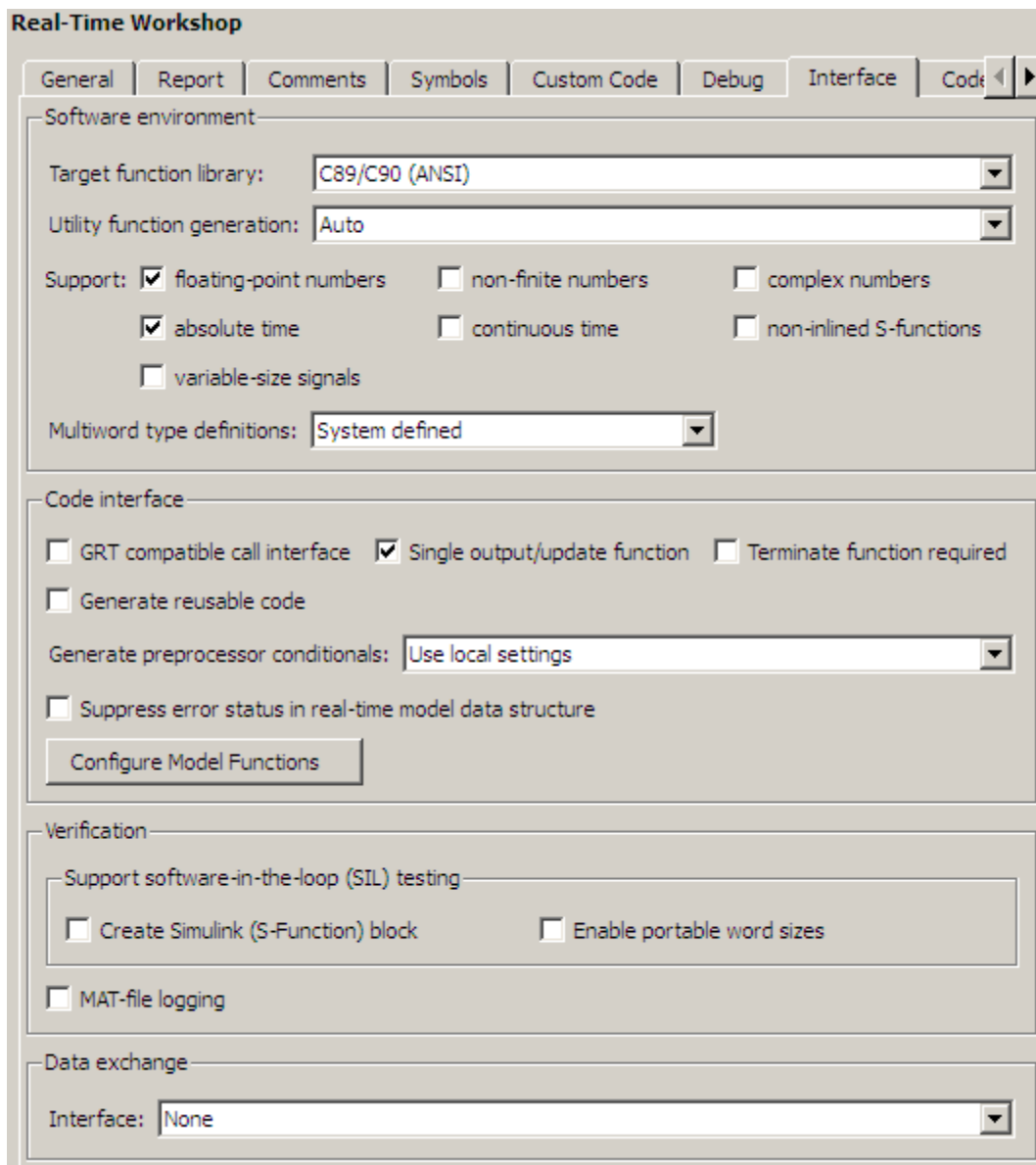
| Application | Setting |
|-------------------|-----------|
| Debugging | On |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | On |

Real-Time Workshop Pane: Interface

The Real-Time Workshop Interface pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select a GRT-based target.

The screenshot shows the 'Real-Time Workshop' dialog box with the 'Interface' tab selected. The dialog has several tabs: General, Report, Comments, Symbols, Custom Code, Debug, and Interface. The 'Interface' tab contains three sections: 'Software environment', 'Verification', and 'Data exchange'. The 'Software environment' section has two dropdown menus: 'Target function library' set to 'C89/C90 (ANSI)' and 'Utility function generation' set to 'Auto'. The 'Verification' section has a dropdown menu for 'MAT-file variable name modifier' set to 'rt_'. The 'Data exchange' section has a dropdown menu for 'Interface' set to 'None'. At the bottom of the dialog, there is a checkbox for 'Generate code only' which is unchecked, and a 'Build' button. Below the dialog, there are three buttons: 'Revert', 'Help', and 'Apply'. A help icon (question mark in a circle) is located at the bottom left of the dialog.

The Real-Time Workshop Interface pane includes the following parameters when the Real-Time Workshop product is installed on your system and you select an ERT-based target. ERT-based target parameters require a Real-Time Workshop Embedded Coder license when generating code.



In this section...

- “Real-Time Workshop: Interface Tab Overview” on page 6-145
- “Target function library” on page 6-146
- “Utility function generation” on page 6-148
- “Support: floating-point numbers” on page 6-150
- “Support: absolute time” on page 6-152
- “Support: non-finite numbers” on page 6-154
- “Support: continuous time” on page 6-156
- “Support: complex numbers” on page 6-158
- “Support: non-inlined S-functions” on page 6-160
- “Support: variable-size signals” on page 6-162
- “Multiword type definitions” on page 6-163
- “Maximum word length” on page 6-165
- “GRT compatible call interface” on page 6-167
- “Single output/update function” on page 6-169
- “Terminate function required” on page 6-171
- “Generate reusable code” on page 6-173
- “Reusable code error diagnostic” on page 6-176
- “Pass root-level I/O as” on page 6-178
- “Block parameter visibility” on page 6-180
- “Internal data visibility” on page 6-182
- “Block parameter access” on page 6-184
- “Internal data access” on page 6-186
- “External I/O access” on page 6-188
- “Generate destructor” on page 6-190
- “Use operator new for referenced model object registration” on page 6-192
- “Generate preprocessor conditionals” on page 6-194

In this section...

“Suppress error status in real-time model data structure” on page 6-196

“Configure Model Functions” on page 6-198

“Configure C++ Encapsulation Interface” on page 6-199

“Create Simulink (S-Function) block” on page 6-200

“Enable portable word sizes” on page 6-202

“MAT-file logging” on page 6-204

“MAT-file variable name modifier” on page 6-206

“Interface” on page 6-208

“Signals in C API” on page 6-211

“Parameters in C API” on page 6-212

“Transport layer” on page 6-213

“MEX-file arguments” on page 6-215

“Static memory allocation” on page 6-217

“Static memory buffer size” on page 6-219

Real-Time Workshop: Interface Tab Overview

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

See Also

- [Configuring Model Interfaces](#)
- [“Real-Time Workshop Pane: Interface” on page 6-141](#)

Target function library

Specify a target-specific math library for your model.

Settings

Default: C89/C90 (ANSI)

C89/C90 (ANSI)

Generates calls to the ISO[®]/IEC 9899:1990 C standard math library for floating-point functions.

C99 (ISO)

Generates calls to the ISO/IEC 9899:1999 C standard math library.

GNU99 (GNU)

Generates calls to the GNU[®] gcc math library, which provides C99 extensions as defined by compiler option `-std=gnu99`.

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

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Command-Line Information

Parameter: GenFloatMathFcnCalls

Type: string

Value: 'ANSI_C' | 'C99 (ISO)' | 'GNU99 (GNU)'

Default: 'ANSI_C'

Recommended Settings

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

See Also

Configuring Model Interfaces

Utility function generation

Specify the location for generating utility functions.

Settings

Default: Auto

Auto

Operates as follows:

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

Shared location

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

Command-Line Information

Parameter: UtilityFuncGeneration

Type: string

Value: 'Auto' | 'Shared location'

Default: 'Auto'

Recommended Settings

| Application | Setting |
|-------------------|--|
| Debugging | Shared location (GRT) No impact (ERT) |
| Traceability | Shared location (GRT) No impact (ERT) |
| Efficiency | Shared location |
| Safety precaution | No impact |

See Also

Configuring Model Interfaces

Support: floating-point numbers

Specify whether to generate floating-point data and operations.

Settings

Default: On (GUI), 'off' (command-line)



On

Generates floating-point data and operations.



Off

Generates pure integer code. If you clear this option, an error occurs if the code generator encounters floating-point data or expressions. The error message reports offending blocks and parameters.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Selecting this parameter enables **Support: non-finite numbers** and clearing this parameter disables **Support: non-finite numbers**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: PurelyIntegerCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Note The command-line values are reverse of the settings values. Therefore, 'on' in the command line corresponds to the description of “Off” in the settings section, and 'off' in the command line corresponds to the description of “On” in the settings section.

Recommended Settings

| Application | Setting |
|--------------------|---|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | Off (GUI), 'on' (command-line) — for integer only |
| Safety precaution | No impact |

Support: absolute time

Specify whether to generate and maintain integer counters for absolute and elapsed time values.

Settings

Default: on



On

Generates and maintains integer counters for blocks that require absolute or elapsed time values. Absolute time is the time from the start of program execution to the present time. An example of elapsed time is time elapsed between two trigger events.

If you select this option and the model does not include blocks that use time values, the target does not generate the counters.



Off

Does not generate integer counters to represent absolute or elapsed time values. If you do not select this option and the model includes blocks that require absolute or elapsed time values, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- You must select this parameter if your model includes blocks that require absolute or elapsed time values.

Command-Line Information

Parameter: SupportAbsoluteTime

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

See Also

Timing Services

Support: non-finite numbers

Specify whether to generate nonfinite data and operations.

Settings

Default: on

- On
Generates nonfinite data (for example, NaN and Inf) and related operations.
- Off
Does not generate nonfinite data and operations. If you clear this option, an error occurs if the code generator encounters nonfinite data or expressions. The error message reports offending blocks and parameters.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled by **Support: floating-point numbers**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: SupportNonFinite
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

| Application | Setting |
|-------------|-----------|
| Debugging | No impact |

| Application | Setting |
|--------------------|----------------|
| Traceability | No impact |
| Efficiency | Off |
| Safety precaution | Off |

Support: continuous time

Specify whether to generate code for blocks that use continuous time.

Settings

Default: off



On

Generates code for blocks that use continuous time.



Off

Does not generate code for blocks that use continuous time. If you do not select this option and the model includes blocks that use continuous time, an error occurs during code generation.

Dependencies

- This option only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This option must be on if your model includes blocks that require absolute or elapsed time values.
- This option must be off when generating an S-function wrapper for an ERT target; the code generator does not support continuous time for this target scenario.
- If you have customized `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, be aware that selecting the options **Support: continuous time** and **Single output/update function** together may cause output values read from `ert_main` for a continuous output port to differ from the corresponding output values in the model's logged data. This is because, while logged data is a snapshot of output at major time steps, output read from `ert_main` after the base-rate model step potentially reflects intervening minor time steps. To eliminate the discrepancy, either separate the generated output and update functions (clear the **Single output/update function** option) or place a Zero-Order Hold block before the continuous output port.

Command-Line Information

Parameter: SupportContinuousTime

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

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

Support: complex numbers

Specify whether to generate complex data and operations.

Settings

Default: on



On

Generates complex numbers and related operations.



Off

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: SupportComplex

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | No impact |
| Traceability | No impact |

| Application | Setting |
|--------------------|---------------------|
| Efficiency | Off (for real only) |
| Safety precaution | No impact |

Support: non-inlined S-functions

Specify whether to generate code for noninlined S-functions.

Settings

Default: Off



On

Generates code for noninlined S-functions.



Off

Does not generate code for noninlined S-functions. If this parameter is off and the model includes a noninlined S-function, an error occurs during the build process.

Tip

- Inlining S-functions is highly advantageous in production code generation, for example, for implementing device drivers. In such cases, clear this option to enforce use of inlined S-functions for code generation.
- Noninlined S-functions require additional memory and computation resources, and can result in significant performance issues. Consider using an inlined S-function when efficiency is a concern.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Selecting this parameter also selects **Support: floating-point numbers** and **Support: non-finite numbers**. If you clear **Support: floating-point numbers** or **Support: non-finite numbers**, a warning is displayed during code generation because these parameters are required by the S-function interface.

Command-Line Information

Parameter: SupportNonInlinedSFcns

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

- Automatic S-Function Wrapper Generation
- Writing S-Functions for Real-Time Workshop Code Generation

Support: variable-size signals

Specify whether to generate code for models that use variable-size signals.

Settings

Default: Off



On

Generates code for models that use variable-size signals.



Off

Does not generate code for models that use variable-size signals. If this parameter is off and the model uses variable-size signals, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: SupportVariableSizeSignals

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

Multiword type definitions

Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.

Settings

Default: System defined

System defined

Use the default system type definitions for multiword data types in generated code. During code generation, if multiword usage is detected, multiword types will be generated into the file `rtwtypes.h`.

User defined

Allows you to control how multiword type definitions are handled during the code generation process. Selecting this value enables the associated parameter **Maximum word length**, which allows you to specify a maximum word length, in bits, for which the code generation process will generate multiword types into the file `rtwtypes.h`. The default maximum word length is 256. If you select 0, no multiword types are generated into the file `rtwtypes.h`, which provides you complete control over type definitions for multiword data types in generated code.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Selecting the value `User defined` for this parameter enables the associated parameter **Maximum word length**.

Command-Line Information

Parameter: `ERTMultiwordTypeDef`

Type: `string`

Value: `'System defined' | 'User defined'`

Default: `'System defined'`

Recommended Settings

| Application | Setting |
|-------------------|--|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | Specifying <code>User_defined</code> and a low value for Maximum word length reduces the size of the generated file <code>rtwtypes.h</code> |
| Safety precaution | Use default |

Maximum word length

Specify a maximum word length, in bits, for which the code generation process will generate system-defined multiword types

Settings

Default: 256

Specify a maximum word length, in bits, for which the code generation process will generate multiword types into the file `rtwtypes.h`. All multiword types up to and including this number of bits will be generated. If you select 0, no multiword types are generated into the file `rtwtypes.h`, which provides you complete control over type definitions for multiword data types in generated code.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled by selecting the value `User` defined for the parameter **Multiword type definitions**.

Command-Line Information

Parameter: `ERTMaxMultiwordLength`

Type: integer

Value: Any valid quantity of bits representing a word size

Default: 256

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | No impact |
| Traceability | No impact |

| Application | Setting |
|--------------------|--|
| Efficiency | Smaller values reduce the size of the generated file <code>rtwtypes.h</code> |
| Safety precaution | Use default |

GRT compatible call interface

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

Settings

Default: off



On

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

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



Off

Disables the GRT compatible call interface.

Tips

The following are unsupported:

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

Dependencies

- This parameter only appears for ERT-based targets with **Language** set to C or C++ (not C++ (Encapsulated)).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Selecting this parameter also selects the required option **Support: floating-point numbers**. If you subsequently clear **Support: floating-point numbers**, an error is displayed during code generation.
- Selecting this parameter disables the incompatible option **Single output/update function**. Clearing this parameter enables **Single output/update function**.

Command-Line Information

Parameter: GRTInterface

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

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

Single output/update function

Specify whether to generate the *model_step* function.

Settings

Default: on



On

Generates the *model_step* function for a model. This function contains the output and update function code for all blocks in the model and is called by `rt_OneStep` to execute processing for one clock period of the model at interrupt level.



Off

Does not combine output and update function code for model blocks in a single function, and instead generates the code in separate *model_output* and *model_update* functions.

Tips

Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. See Model Blocks and Direct Feedthrough for details.

Dependencies

- This option only appears for ERT-based targets with **Language** set to C or C++ (not C++ (Encapsulated)).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This option and **GRT compatible call interface** are mutually incompatible and cannot both be selected through the GUI. Selecting **GRT compatible call interface** disables this option and clearing **GRT compatible call interface** enables this option.
- When you use this option, you must clear the option **Minimize algebraic loop occurrences** on the **Model Referencing** pane.
- If you have customized `ert_main.c` or `.cpp` to read model outputs after each base-rate model step, be aware that selecting the options **Support:**

continuous time and **Single output/update function** together may cause output values read from `ert_main` for a continuous output port to differ from the corresponding output values in the model's logged data. This is because, while logged data is a snapshot of output at major time steps, output read from `ert_main` after the base-rate model step potentially reflects intervening minor time steps. To eliminate the discrepancy, either separate the generated output and update functions (clear the **Single output/update function** option) or place a Zero-Order Hold block before the continuous output port.

Command-Line Information

Parameter: `CombineOutputUpdateFcns`

Type: `string`

Value: `'on' | 'off'`

Default: `'on'`

Recommended Settings

| Application | Setting |
|-------------------|---------|
| Debugging | On |
| Traceability | On |
| Efficiency | On |
| Safety precaution | On |

See Also

`rt_OneStep`

Terminate function required

Specify whether to generate the `model_terminate` function.

Settings

Default: on



On

Generates a `model_terminate` function. This function contains all model termination code and should be called as part of system shutdown.



Off

Does not generate a `model_terminate` function. Suppresses the generation of this function if you designed your application to run indefinitely and does not require a terminate function.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: IncludeMdlTerminateFcn

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | No impact |
| Traceability | No impact |

| Application | Setting |
|--------------------|----------------|
| Efficiency | Off |
| Safety precaution | Off |

See Also

`model_terminate`

Generate reusable code

Specify whether to generate reusable, reentrant code.

Settings

Default: off



On

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



Off

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

Tips

- Entry points are exported with *model.h*. To call the entry-point functions from hand-written code, add an `#include model.h` directive to the code. If this option is selected, you must examine the generated code to determine the calling interface required for these functions.
- When this option is selected, the code generator generates a pointer to the real-time model object (*model_M*).
- In some cases, when this option is selected, the code generator might generate code that compiles but is not reentrant. For example, if any signal, DWork structure, or parameter data has a storage class other than Auto, global data structures are generated.

Dependencies

- This parameter only appears for ERT-based targets with **Language** set to C or C++ (not C++ (Encapsulated)).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter enables **Reusable code error diagnostic** and **Pass root-level I/O as**.
- You must clear this option if you are using:
 - The static `ert_main.c` module, rather than generating a main program
 - The `model_step` function prototype control capability
 - The subsystem parameter **Function with separate data**
 - A subsystem that
 - Has multiple ports that share the same source
 - Has a port used by multiple instances has different sample times, data types, complexity, frame status, or dimension across the instances
 - Has output marked as a global signal
 - For each instance contains identical blocks with different names or parameter settings
- This parameter has no effect on code generated for function-call subsystems.

Command-Line Information

Parameter: MultiInstanceERTCode

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

| Application | Setting |
|-------------|-----------|
| Debugging | No impact |

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

See Also

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

Reusable code error diagnostic

Select the severity level for diagnostics displayed when a model violates requirements for generating reusable code.

Settings

Default: Error

None

Proceed with build without displaying a diagnostic message.

Warning

Proceed with build after displaying a warning message.

Error

Abort build after displaying an error message.

Under certain conditions, the Real-Time Workshop Embedded Coder software might

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

Dependencies

- This parameter only appears for ERT-based targets with **Language** set to C or C++ (not C++ (Encapsulated)).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled by **Generate reusable code**.

Command-Line Information

Parameter: MultiInstanceErrorCode

Type: string

Value: 'None' | 'Warning' | 'Error'

Default: 'Error'

Recommended Settings

| Application | Setting |
|-------------------|------------------|
| Debugging | Warning or Error |
| Traceability | No impact |
| Efficiency | None |
| Safety precaution | No impact |

See Also

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

Pass root-level I/O as

Control how root-level model input and output are passed to the `model_step` function.

Settings

Default: Individual arguments

Individual arguments

Passes each root-level model input and output value to `model_step` as a separate argument.

Structure reference

Packs all root-level model input into a struct and passes struct to `model_step` as an argument. Similarly, packs root-level model output into a second struct and passes it to `model_step`.

Dependencies

- This parameter only appears for ERT-based targets with **Language** set to C or C++ (not C++ (Encapsulated)).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is enabled by **Generate reusable code**.

Command-Line Information

Parameter: RootIOFormat

Type: string

Value: 'Individual arguments' | 'Structure reference'

Default: 'Individual arguments'

Recommended Settings

| Application | Setting |
|-------------|-----------|
| Debugging | No impact |

| Application | Setting |
|--------------------|----------------|
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

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

Block parameter visibility

Specify whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class.

Settings

Default: `private`

`public`

Generates the block parameter structure as a `public` data member of the C++ model class.

`private`

Generates the block parameter structure as a `private` data member of the C++ model class.

`protected`

Generates the block parameter structure as a `protected` data member of the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: `ParameterMemberVisibility`

Type: `string`

Value: `'public' | 'private' | 'protected'`

Default: `'private'`

Recommended Settings

| Application | Setting |
|-------------|-----------|
| Debugging | No impact |

| Application | Setting |
|--------------------|----------------|
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | protected |

See Also

“Configuring Code Interface Options”

Internal data visibility

Specify whether to generate internal data structures such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states as public, private, or protected data members of the C++ model class.

Settings

Default: private

public

Generates internal data structures as public data members of the C++ model class.

private

Generates internal data structures as private data members of the C++ model class.

protected

Generates internal data structures as protected data members of the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: InternalMemberVisibility

Type: string

Value: 'public' | 'private' | 'protected'

Default: 'private'

Recommended Settings

| Application | Setting |
|-------------------|-----------|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | protected |

See Also

“Configuring Code Interface Options”

Block parameter access

Specify whether to generate access methods for block parameters for the C++ model class.

Settings

Default: None

None

Does not generate access methods for block parameters for the C++ model class.

Method

Generates noninlined access methods for block parameters for the C++ model class.

Inlined method

Generates inlined access methods for block parameters for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateParameterAccessMethods

Type: string

Value: 'None' | 'Method' | 'Inlined method'

Default: 'None'

Recommended Settings

| Application | Setting |
|-------------|----------------|
| Debugging | Inlined method |

| Application | Setting |
|--------------------|----------------|
| Traceability | Inlined method |
| Efficiency | Inlined method |
| Safety precaution | None |

See Also

“Configuring Code Interface Options”

Internal data access

Specify whether to generate access methods for internal data structures, such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states, for the C++ model class.

Settings

Default: None

None

Does not generate access methods for internal data structures for the C++ model class.

Method

Generates noninlined access methods for internal data structures for the C++ model class.

Inlined method

Generates inlined access methods for internal data structures for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateInternalMemberAccessMethods

Type: string

Value: 'None' | 'Method' | 'Inlined method'

Default: 'None'

Recommended Settings

| Application | Setting |
|-------------------|----------------|
| Debugging | Inlined method |
| Traceability | Inlined method |
| Efficiency | Inlined method |
| Safety precaution | None |

See Also

“Configuring Code Interface Options”

External I/O access

Specify whether to generate access methods for root-level I/O signals for the C++ model class.

Note This parameter affects generated code only if you are using the default (void-void style) step method for your model class; *not* if you are explicitly passing arguments for root-level I/O signals using an I/O arguments style step method. For more information, see “Passing No Arguments (void-void)” and “Passing I/O Arguments”.

Settings

Default: None

None

Does not generate access methods for root-level I/O signals for the C++ model class.

Method

Generates noninlined access methods for root-level I/O signals for the C++ model class.

Inlined method

Generates inlined access methods for root-level I/O signals for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateExternalIOAccessMethods

Type: string

Value: 'None' | 'Method' | 'Inlined method'

Default: 'None'

Recommended Settings

| Application | Setting |
|-------------------|----------------|
| Debugging | Inlined method |
| Traceability | Inlined method |
| Efficiency | Inlined method |
| Safety precaution | None |

See Also

“Configuring Code Interface Options”

Generate destructor

Specify whether to generate a destructor for the C++ model class.

Settings

Default: on

On
Generates a destructor for the C++ model class.

Off
Does not generate a destructor for the C++ model class.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateDestructor

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

See Also

“Configuring Code Interface Options”

Use operator new for referenced model object registration

Specify whether generated code uses the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ encapsulation interface.

Settings

Default: off



On

Generates code that uses dynamic memory allocation to instantiate objects for referenced models configured with a C++ encapsulation interface. Specifically, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses `new` to instantiate objects for referenced models.

Selecting this option frees a parent model from having to maintain information about submodels beyond its direct children.



Off

Does not generate code that uses `new` to instantiate referenced model objects.

Clearing this option means that a parent model maintains information about all of its submodels, including its direct and indirect children.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: UseOperatorNewForModelRefRegistration

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

“Configuring Code Interface Options”

Generate preprocessor conditionals

Generate preprocessor conditional directives globally for a model or locally for each Model block with variant models.

Settings

Default: Use local settings

Use local settings

Generates preprocessor conditional directives based on the value of the **Generate preprocessor conditionals** parameter on the Model block parameters dialog. If you select the **Generate preprocessor conditionals** parameter in the Model block parameters dialog, the generated code contains preprocessor conditional directives for all variant models of that Model block. If you do not select this parameter for a Model block, code is generated for the active variant model.

Enable all

Generates preprocessor conditional directives for all variant models of the Model blocks. Disables the **Generate preprocessor conditionals** parameter in the Model block parameters dialog.

Disable all

Only generates code for the active variant model of the Model block. Disables the **Generate preprocessor conditionals** parameter in the Model block parameters dialog for all Model blocks.

Tips

For generating preprocessor directives we recommend the following settings:

- Select the “Inline parameters” parameter on the Optimization pane of the Configuration Parameters dialog box.
- Deselect the “Ignore custom storage classes” on page 6-21 parameter on the Real-Time Workshop pane of the Configuration Parameters dialog.

Dependencies

- This parameter only appears for ERT-based targets.

- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- Setting this parameter to `Use local settings` enables **Generate preprocessor conditionals** parameter on the Model block parameters dialog.
- Setting this parameter to `Enable all` or `Disable all` disables the **Generate preprocessor conditionals** check box on the Model block parameters dialog.
- Setting this parameter to `Enable all` sets the **Selected variant** control on the Model block parameter dialog to `(derive from conditions)`.

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`

Type: `string`

Value: `'Use local settings' | 'Enable all' | 'Disable all'`

Default: `'Use local settings'`

Recommended Settings

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

See Also

- “Using Model Reference Variants”
- “Generating Code Variants for Variant Models”

Suppress error status in real-time model data structure

Specify whether to log or monitor error status.

Settings

Default: off



On

Omits the error status field from the generated real-time model data structure `rtModel`. This option reduces memory usage.

Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.



Off

Includes an error status field in the generated real-time model data structure `rtModel`. You can use available macros to monitor the field for error message data or set it with error message data.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- This parameter is cleared if you select the incompatible option **MAT-file logging**. If you subsequently select this parameter, code generation displays an error.
- Selecting this parameter clears **Support: continuous time**.
- If your application contains multiple integrated models, the setting of this option must be the same for all of the models to avoid unexpected application behavior. For example, if you select the option for one model but not another, an error status might not get registered by the integrated application.

Command-Line Information

Parameter: SuppressErrorStatus

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

rtModel Accessor Macros

Configure Model Functions

Click the **Configure Model Functions** button to open the Model Interface dialog box. In this dialog box, you can specify whether the code generator uses default `model_initialize` and `model_step` function prototypes or model-specific C prototypes. Based on your selection, you can preview and modify the function prototypes.

Dependency

- This parameter appears only for ERT-based targets with **Language** set to C or C++ (not C++ (Encapsulated)).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

See Also

- Controlling Model Function Prototypes
- `model_initialize`
- `model_step`
- Launching the Model Interface Dialog Boxes

Configure C++ Encapsulation Interface

Click the **Configure C++ Encapsulation Interface** button to open the Configure C++ encapsulation interface dialog box. In this dialog box, you can customize the C++ class interface for your model code. Based on your selections, you can preview and modify the model-specific C++ encapsulation interface.

Dependency

- This parameter appears only for ERT-based targets with **Language** set to C++ (Encapsulated).
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

See Also

- “Controlling Generation of Encapsulated C++ Model Interfaces”
- `model_step`
- “Configuring the Step Method for Your Model Class”

Create Simulink (S-Function) block

Specify whether to generate an S-function block.

Settings

Default: off



On

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

When this option is selected, the Real-Time Workshop software:

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



Off

Does not generate an S-function block.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

Command-Line Information

Parameter: GenerateErtSFunction

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

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

Enable portable word sizes

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

Settings

Default: off



On

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



Off

Does not generate portable code.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.
- When you use this parameter, you should:
 - Select **Create Simulink (S-Function) block**
 - Set **Emulation hardware** on the **Hardware Implementation** pane to None

Command-Line Information

Parameter: PortableWordSizes

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

- Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes
- Tips for Optimizing the Generated Code

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: off



On

Enables MAT-file logging. When you select this option, the generated code saves to MAT-files any data specified in the **Configuration Parameters > Data Import/Export Pane > Save to workspace** subpane, and the data specified by any To Workspace blocks. See “Data Import/Export Pane” and To Workspace. In simulation, this data would be written to the MATLAB workspace, as described in “Exporting Data to the MATLAB Workspace”, but setting MAT-file logging redirects the data to a MAT-file instead. The file is named *model.mat*, where *model* is the name of your model.



Off

Disables MAT-file logging. Clearing this option has the following benefits:

- Eliminates overhead associated with supporting a file system, which typically is not needed for embedded applications
- Eliminates extra code and memory usage required to initialize, update, and clean up logging variables
- Under certain conditions, eliminates code and storage associated with root output ports
- Omits the comparison between the current time and stop time in the *model_step*, allowing the generated program to run indefinitely, regardless of the stop time setting

Dependencies

- This parameter only appears for ERT-based targets and the Tornado® target.
- This parameter requires a Real-Time Workshop Embedded Coder license when generating code.

- Selecting this parameter also selects the required options **Support: floating-point numbers**, **Support: non-finite numbers**, and **Terminate function required**. If you subsequently clear **Support: floating-point numbers**, **Support: non-finite numbers**, or **Terminate function required**, an error is displayed during code generation.
- Selecting this parameter clears the incompatible option **Suppress error status in real-time model data structure**. If you subsequently select **Suppress error status in real-time model data structure**, an error is displayed during code generation.
- Selecting this parameter enables **MAT-file variable name modifier**.
- Clear this option if you are using exported function calls.

Limitation

MAT-file logging does not work in a referenced model, and no code is generated to implement it.

Command-Line Information

Parameter: MatFileLogging

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

Using Virtualized Output Ports Optimization

MAT-file variable name modifier

Select the string to add to MAT-file variable names.

Settings

Default: rt_

rt_ Adds a prefix string.

_rt Adds a suffix string.

none Does not add a string.

Dependency

When an ERT target is selected, this parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: LogVarNameModifier

Type: string

Value: 'none' | 'rt_' | '_rt'

Default: 'rt_'

Recommended Settings

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

See Also

Data Logging

Interface

Specify the data exchange interface (API) to include.

Settings

Default: None

None

Does not include an API in the generated code.

C API

Uses the C API data interface.

External mode

Uses an external data interface.

ASAP2

Uses the ASAP2 data interface.

Dependencies

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

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

Selecting **External mode** enables the following parameters:

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

Command-Line Information

Parameter: see table

Type: string

Value: 'on' | 'off'

Default: 'off'

| To enable... | Set this parameter... | To this value... |
|---------------|--|------------------|
| none | RTWCAPIParams, RTWCAPISignals, ExtMode, GenerateASAP2 | 'off' |
| C API | RTWCAPIParams, RTWCAPISignals | 'on' |
| External mode | ExtMode | 'on' |
| ASAP2 | GenerateASAP2 | 'on' |

Recommended Settings

| Application | Setting |
|-------------------|---|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact during development None for production code generation |

See Also

- “Interacting with Target Application Signals and Parameters Using the C API”
- “Communicating With Code Executing on a Target System Using Simulink External Mode”
- “Using External Mode with the ERT Target”

- “Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration”

Signals in C API

Generate a C API signal structure.

Settings

Default: on



On

Generates C API for global block outputs.



Off

Does not generate C API signals.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information

Parameter: RTWCAPISignals

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

C API for Interfacing with Signals and Parameters

Parameters in C API

Generate C API parameter tuning structures.

Settings

Default: on

- On
Generates C API for global block and model parameters.
- Off
Does not generate C API parameters.

Dependency

This parameter is enabled by selecting **Interface** > C API.

Command-Line Information

Parameter: RTWCAPIParams

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

C API for Interfacing with Signals and Parameters

Transport layer

Specify the transport protocol for external mode communications.

Settings

Default: tcpip

tcpip

Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

serial_win32

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

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. The value is specified either in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`, for targets provided by The MathWorks™, or in an `sl_customization.m` file, for custom targets and/or custom external mode transports.

Dependency

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

Command-Line Information

Parameter: ExtModeTransport

Type: integer

Value: 0 | 1

Default: 0

Recommended Settings

| Application | Setting |
|-------------|-----------|
| Debugging | No impact |

| Application | Setting |
|--------------------|----------------|
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

- Target Interfacing
- Creating an External Mode Communication Channel

MEX-file arguments

Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.

Settings

Default: " "

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

- Network name of your target (for example, 'myputer' or '148.27.151.12')
- Verbosity level (0 for no information or 1 for detailed information)
- TCP/IP server port number (an integer value between 256 and 65535, with a default of 17725)

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

- Verbosity level (0 for no information or 1 for detailed information)
- Serial port ID (for example, 1 for COM1, and so on)
- Baud rate (selected from the set 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, with a default baud rate of 57600)

Dependency

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

Command-Line Information

Parameter: ExtModeMexArgs

Type: string

Value: any valid arguments

Default: " "

Recommended Settings

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

See Also

- Target Interfacing
- Client/Server Implementations

Static memory allocation

Control memory buffer for external mode communication.

Settings

Default: off



On

Enables the **Static memory buffer size** parameter for allocating dynamic memory.



Off

Uses a static memory buffer for external mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- Depending on the specified target, this parameter is enabled by **Data Exchange > Interface > External mode** or by **External Mode**.
- This parameter enables **Static memory buffer size**.

Command-Line Information

Parameter: ExtModeStaticAlloc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|-------------|-----------|
| Debugging | No impact |

| Application | Setting |
|--------------------|----------------|
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

External Mode Interface Options

Static memory buffer size

Specify the memory buffer size for external mode communication.

Settings

Default: 1000000

Enter the number of bytes to preallocate for external mode communications buffers in the target.

Tips

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

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information

Parameter: ExtModeStaticAllocSize

Type: integer

Value: any valid value

Default: 1000000

Recommended Settings

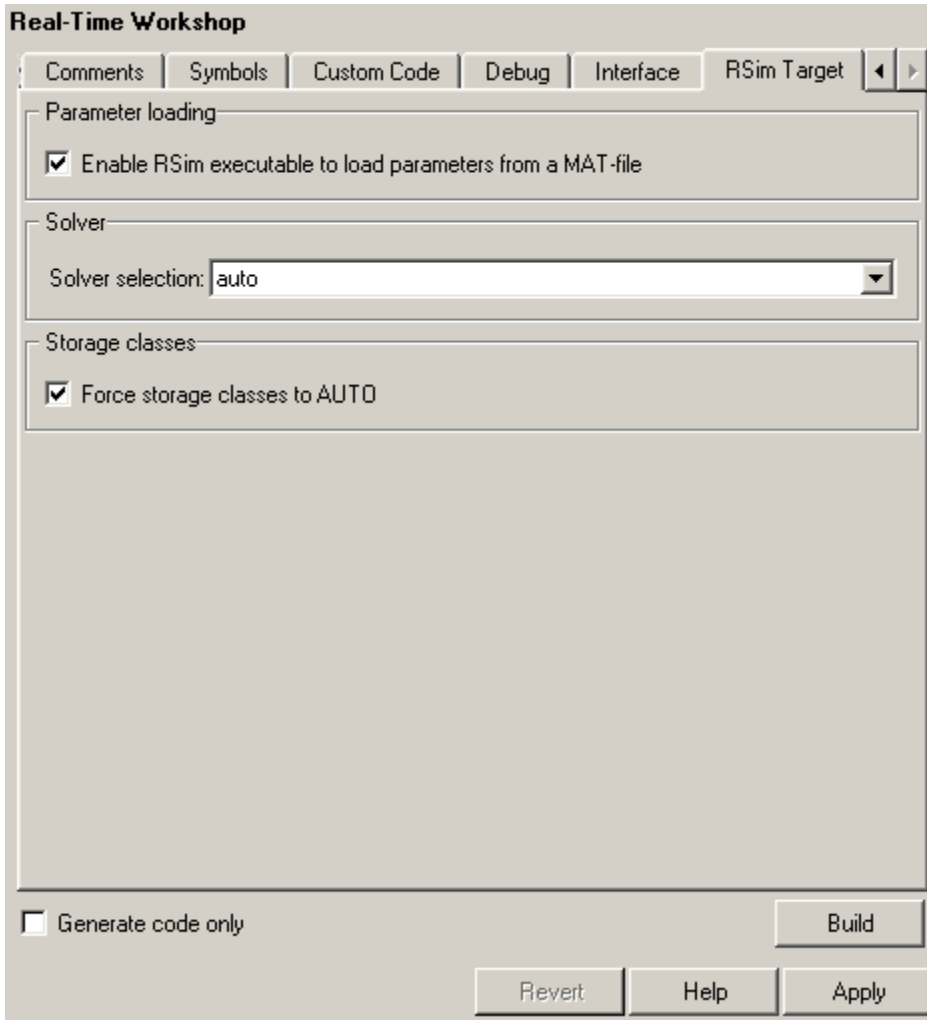
| Application | Setting |
|-------------------|-----------|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

External Mode Interface Options

Real-Time Workshop Pane: RSim Target

The Real-Time Workshop RSim Target pane includes the following parameters when the Real-Time Workshop product is installed on your system and you specify the `rsim.tlc` system target file.



In this section...

“Real-Time Workshop: RSim Target Tab Overview” on page 6-223

“Enable RSim executable to load parameters from a MAT-file” on page 6-224

“Solver selection” on page 6-225

“Force storage classes to AUTO” on page 6-226

Real-Time Workshop: RSim Target Tab Overview

Set configuration parameters for rapid simulation.

Configuration

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

See Also

- [Configuring and Building a Model for Rapid Simulation](#)
- [Running Rapid Simulations](#)
- [“Real-Time Workshop Pane: RSim Target” on page 6-221](#)

Enable RSim executable to load parameters from a MAT-file

Specify whether to load RSim parameters from a MAT-file.

Settings

Default: on



On

Enables RSim to load parameters from a MAT-file.



Off

Disables RSim from loading parameters from a MAT-file.

Command-Line Information

Parameter: RSIM_PARAMETER_LOADING

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

See Also

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

Solver selection

Instruct the target how to select the solver.

Settings

Default: auto

auto

Lets the target choose the solver. The target uses the Simulink solver module if you specify a variable-step solver on the Solver pane. Otherwise, the target uses a Real-Time Workshop built-in solver.

Use Simulink solver module

Instructs the target to use the variable-step solver that you specify on the Solver pane.

Use Real-Time Workshop fixed-step solvers

Instructs the target to use the fixed-step solver that you specify on the Solver pane.

Command-Line Information

Parameter: RSIM_SOLVER_SELECTION

Type: string

Value: 'auto' | 'usesolvermodule' | 'usefixstep'

Default: 'auto'

Recommended Settings

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

Force storage classes to AUTO

Specify whether to retain your storage class settings in a model or to use the automatic settings.

Settings

Default: on



On

Forces the Simulink software to determine all storage classes.



Off

Causes the model to retain storage class settings.

Tips

- Turn this parameter on for flexible custom code interfacing.
- Turn this parameter off when it is necessary to retain storage class settings such as ExportedGlobal or ImportExtern.

Command-Line Information

Parameter: RSIM_STORAGE_CLASS_AUTO

Type: string

Value: 'on' | 'off'

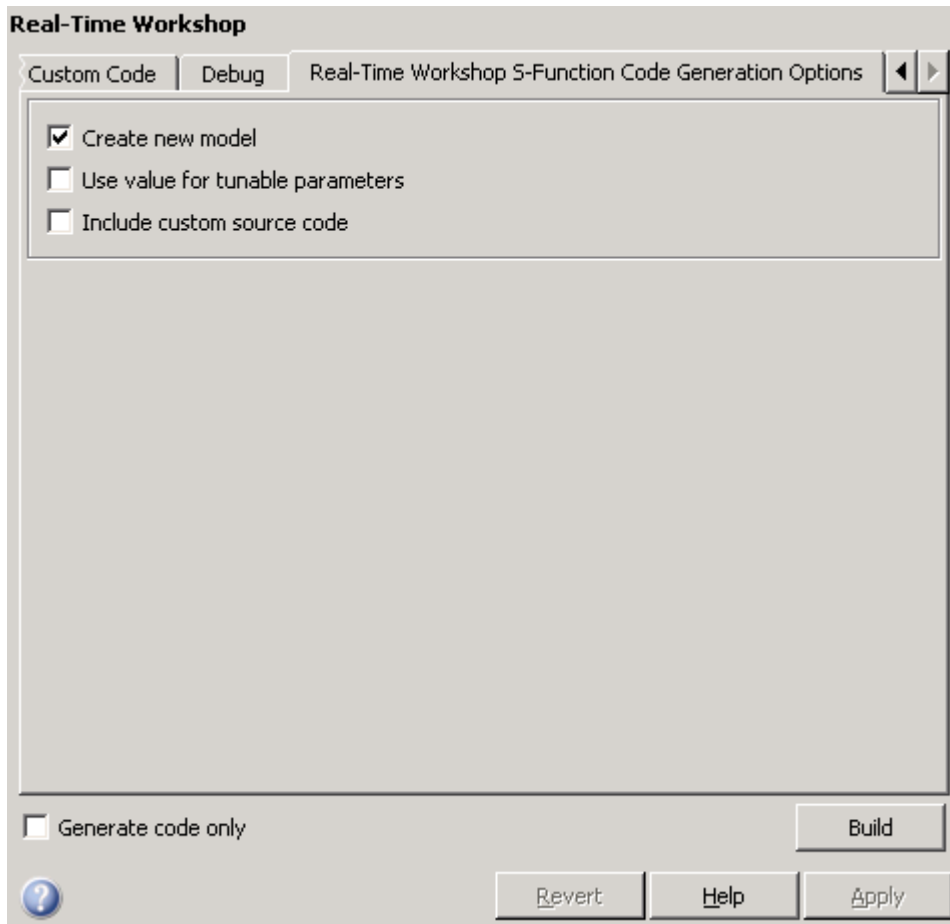
Default: 'on'

Recommended Settings

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

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

The Real-Time Workshop S-Function Code Generation Options pane includes the following parameters when the Real-Time Workshop product is installed on your system and you specify the `rtwsfcn.tlc` system target file.



In this section...

“Real-Time Workshop S-Function Code Generation Options Tab Overview”
on page 6-229

“Create new model” on page 6-230

“Use value for tunable parameters” on page 6-231

“Include custom source code” on page 6-232

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

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

Configuration

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

See Also

- Real-Time Workshop S-Function Code Generation Options
- S-Function Target
- “Real-Time Workshop Pane: Real-Time Workshop S-Function Code Generation Options” on page 6-227

Create new model

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

Settings

Default: on



On

Creates a new model, separate from the current model, containing the generated Real-Time Workshop S-function block.



Off

Generates code but a new model is not created.

Command-Line Information

Parameter: CreateModel

Type: string

Value: 'on' | 'off'

Default: 'on'

See Also

S-Function Target

Use value for tunable parameters

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

Settings

Default: off



On

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



Off

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

Command-Line Information

Parameter: UseParamValues

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

S-Function Target

Include custom source code

Include custom source code in the code generated for the Real-Time Workshop S-function.

Settings

Default: off



On

Always include provided custom source code in the code generated for the Real-Time Workshop S-function.



Off

Do not include custom source code in the code generated for the Real-Time Workshop S-function.

Command-Line Information

Parameter: AlwaysIncludeCustomSrc

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

S-Function Target

Real-Time Workshop Pane: Tornado Target

The Real-Time Workshop Tornado Target pane includes the following parameters when the Real-Time Workshop product is installed on your system and you specify the `tornado.tlc` system target file.

The image shows a screenshot of the 'Real-Time Workshop' configuration window, specifically the 'Tornado Target' tab. The window has a title bar 'Real-Time Workshop' and several tabs: 'General', 'Comments', 'Symbols', 'Custom Code', 'Debug', and 'Tornado Target'. The 'Tornado Target' tab is active. The configuration is organized into several sections:

- Software environment:**
 - Target Function Library: C89/C90 (ANSI) (dropdown menu)
 - Utility function generation: Auto (dropdown menu)
- Tornado:**
 - MAT-file logging
 - MAT-file variable name modifier: rt_ (dropdown menu)
 - Code Format: RealTime (dropdown menu)
 - StethoScope
 - Download to VxWorks target
- VxWorks:**
 - Base task priority: 30 (text input field)
 - Task stack size: 16384 (text input field)
- External mode options:**
 - External mode

At the bottom of the window, there is a checkbox for 'Generate code only' and a 'Build' button. Below these are three buttons: 'Revert', 'Help', and 'Apply'.

In this section...

“Real-Time Workshop: Tornado Target Tab Overview” on page 6-235

“Target function library” on page 6-236

“Utility function generation” on page 6-238

“MAT-file logging” on page 6-239

“MAT-file variable name modifier” on page 6-241

“Code Format” on page 6-243

“StethoScope” on page 6-244

“Download to VxWorks target” on page 6-246

“Base task priority” on page 6-248

“Task stack size” on page 6-250

“External mode” on page 6-251

“Transport layer” on page 6-253

“MEX-file arguments” on page 6-255

“Static memory allocation” on page 6-257

“Static memory buffer size” on page 6-259

Real-Time Workshop: Tornado Target Tab Overview

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

Configuration

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

See Also

- *Tornado User's Guide* from Wind River® Systems
- *StethoScope User's Guide* from Wind River Systems
- Targeting Tornado for Real-Time Applications
- “Real-Time Workshop Pane: Tornado Target” on page 6-233

Target function library

Specify a target-specific math library for your model.

Settings

Default: C89/C90 (ANSI)

C89/C90 (ANSI)

Generates calls to the ISO/IEC 9899:1990 C standard math library for floating-point functions.

C99 (ISO)

Generates calls to the ISO/IEC 9899:1999 C standard math library.

GNU99 (GNU)

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

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Command-Line Information

Parameter: GenFloatMathFcnCalls

Type: string

Value: 'ANSI_C' | 'C99 (ISO)' | 'GNU99 (GNU)'

Default: 'ANSI_C'

Recommended Settings

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

See Also

Configuring Model Interfaces

Utility function generation

Specify the location for generating utility functions.

Settings

Default: Auto

Auto

Operates as follows:

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

Shared location

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

Command-Line Information

Parameter: UtilityFuncGeneration

Type: string

Value: 'Auto' | 'Shared location'

Default: 'Auto'

Recommended Settings

| Application | Setting |
|-------------------|-----------------|
| Debugging | Shared location |
| Traceability | Shared location |
| Efficiency | Shared location |
| Safety precaution | No impact |

See Also

Configuring Model Interfaces

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: off



On

Enables MAT-file logging. When you select this option, the generated code saves to MAT-files any data specified in the **Configuration Parameters > Data Import/Export Pane > Save to workspace** subpane, and the data specified by any To Workspace blocks. See “Data Import/Export Pane” and To Workspace. In simulation, this data would be written to the MATLAB workspace, as described in “Exporting Data to the MATLAB Workspace”, but setting MAT-file logging redirects the data to a MAT-file instead. The file is named *model.mat*, where *model* is the name of your model.



Off

Disables MAT-file logging. Clearing this option has the following benefits:

- Eliminates overhead associated with supporting a file system, which typically is not needed for embedded applications
- Eliminates extra code and memory usage required to initialize, update, and clean up logging variables
- Under certain conditions, eliminates code and storage associated with root output ports
- Omits the comparison between the current time and stop time in the *model_step*, allowing the generated program to run indefinitely, regardless of the stop time setting

Dependencies

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

Limitation

MAT-file logging does not work in a referenced model, and no code is generated to implement it.

Command-Line Information

Parameter: MatFileLogging

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

Using Virtualized Output Ports Optimization

MAT-file variable name modifier

Select the string to add to the MAT-file variable names.

Settings

Default: rt_

rt_ Adds a prefix string.

_rt Adds a suffix string.

none Does not add a string.

Dependency

When an ERT target is selected, this parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: LogVarNameModifier

Type: string

Value: 'none' | 'rt_' | '_rt'

Default: 'rt_'

Recommended Settings

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

See Also
Data Logging

Code Format

Specify the code generation format.

Settings

Default: RealTime

RealTime

Specifies the Real-Time code generation format.

RealTimeMalloc

Specifies the Real-Time Malloc code generation format.

Command-Line Information

Parameter: CodeFormat

Type: string

Value: 'RealTime' | 'RealTimeMalloc'

Default: 'RealTime'

Recommended Settings

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

See Also

Targeting Tornado for Real-Time Applications

StethoScope

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

Settings

Default: off



On

Enables StethoScope.



Off

Disables StethoScope.

Tips

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

Dependencies

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

Command-Line Information

Parameter: StethoScope

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | On |
| Traceability | No impact |

| Application | Setting |
|--------------------|----------------|
| Efficiency | Off |
| Safety precaution | Off |

See Also

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

Download to VxWorks target

Specify whether to automatically download the generated program to the VxWorks target.

Settings

Default: off



On

Automatically downloads the generated program to VxWorks after each build.



Off

Does not automatically download to VxWorks, you must download generated programs manually.

Tips

- Automatic download requires specifying the target name and host name in the makefile, as described in [Configuring for Automatic Downloading](#).
- Before every build, reset VxWorks by pressing **Ctrl+X** on the host console or power-cycling the VxWorks chassis. This ensures that no dangling processes or stale data exist in VxWorks when the automatic download occurs.

Command-Line Information

Parameter: DownloadToVxWorks

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | No impact |
| Traceability | No impact |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | Off |

See Also

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

Base task priority

Specify the priority with which the base rate task for the model is to be spawned.

Settings

Default: 30

Tips

- For a multirate, multitasking model, the Real-Time Workshop software increments the priority of each subrate task by one.
- The value you specify for this option will be overridden by a base priority specified in a call to the `rt_main()` function spawned as a task.

Command-Line Information

Parameter: BasePriority
Type: integer
Value: any valid value
Default: 30

Recommended Settings

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

See Also

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

- Targeting Tornado for Real-Time Applications

Task stack size

Stack size in bytes for each task that executes the model.

Settings

Default: 16384

Command-Line Information

Parameter: TaskStackSize

Type: integer

Value: any valid value

Default: 16384

Recommended Settings

| Application | Setting |
|-------------------|--|
| Debugging | No impact |
| Traceability | No impact |
| Efficiency | Larger stack may waste space |
| Safety precaution | Larger stack reduces the possibility of overflow |

See Also

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

External mode

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

Settings

Default: on



On

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



Off

Disables external mode.

Dependencies

Selecting this parameter enables:

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

Command-Line Information

Parameter: ExtMode

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

| Application | Setting |
|--------------|-----------|
| Debugging | No impact |
| Traceability | No impact |

| Application | Setting |
|--------------------|----------------|
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

External Mode

Transport layer

Specify the transport protocol for external mode communications.

Settings

Default: tcpip

tcpip

Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. For targets provided by The MathWorks, the value is specified in `matlabroot/toolbox/simulink/simulink/extmode_transports.m`.

Dependency

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

Command-Line Information

Parameter: ExtModeTransport

Type: integer

Value: 0 | 1

Default: 0

Recommended Settings

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

See Also

Target Interfacing

MEX-file arguments

Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.

Settings

Default: " "

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

- Network name of your target (for example, 'myPutter' or '148.27.151.12')
- Verbosity level (0 for no information or 1 for detailed information)
- TCP/IP server port number (an integer value between 256 and 65535, with a default of 17725)

Dependency

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

Command-Line Information

Parameter: ExtModeMexArgs

Type: string

Value: any valid arguments

Default: " "

Recommended Settings

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

See Also

- Target Interfacing
- Client/Server Implementations

Static memory allocation

Control the memory buffer for external mode communication.

Settings

Default: off



On

Enables the **Static memory buffer size** parameter for allocating dynamic memory.



Off

Uses a static memory buffer for external mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- This parameter is enabled by **External Mode**.
- This parameter enables **Static memory buffer size**.

Command-Line Information

Parameter: ExtModeStaticAlloc

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

| Application | Setting |
|-------------|-----------|
| Debugging | No impact |

| Application | Setting |
|--------------------|----------------|
| Traceability | No impact |
| Efficiency | No impact |
| Safety precaution | No impact |

See Also

External Mode Interface Options

Static memory buffer size

Specify the memory buffer size for external mode communication.

Settings

Default: 1000000

Enter the number of bytes to preallocate for external mode communications buffers in the target.

Tips

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

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information

Parameter: ExtModeStaticAllocSize

Type: integer

Value: any valid value

Default: 1000000

Recommended Settings

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

See Also

External Mode Interface Options

Parameter Reference

In this section...

“Recommended Settings Summary” on page 6-261

“Parameter Command-Line Information Summary” on page 6-285

Recommended Settings Summary

The following table summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the GRT and ERT targets, unless otherwise specified.

For parameters that are available only when an ERT target is specified, see the “Recommended Settings Summary” in the Real-Time Workshop Embedded Coder documentation.

For additional details, click the links in the Configuration Parameter column.

Mapping Application Requirements to the Solver Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------|----------------------------|--------------|------------|--------------------|---|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Start Time | No impact | No impact | No impact | 0.0 | 0.0 seconds |
| Stop time | No impact | No impact | No impact | Any positive value | 10.0 seconds |
| Type | Fixed-step | Fixed-step | Fixed-step | Fixed-step | Variable-step (you must change to Fixed-step for code generation) |

Mapping Application Requirements to the Solver Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-----------------------------------|----------------------------|--------------|------------|---|-------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Solver” | No impact | No impact | No impact | Discrete (no continuous states) | ode3 (Bogacki-Shampine) |
| “Periodic sample time constraint” | No impact | No impact | No impact | Specified or Ensure sample time independent | Unconstrained |
| “Sample time properties” | No impact | No impact | No impact | Period, offset, and priority of each sample time in the model; faster sample times must have higher priority than slower sample times | ' ' |

Mapping Application Requirements to the Solver Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|-----------------------------------|---|-------------------|--------------------------|------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Tasking mode for periodic sample times | No impact | No impact | No impact | No impact | Auto |
| “Automatically handle rate transition for data transfer” | No impact | No impact (for simulation and during development) Off (for production code generation) | No impact | Off | Off |

Mapping Application Requirements to the Data Import/Export Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--------------------------------|-----------------------------------|---------------------|-------------------|------------------------------|------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Input” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | Off |
| “Initial State” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | Off |

Mapping Application Requirements to the Data Import/Export Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|------------------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Time” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | On |
| “States” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | Off |
| “Output” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | On |
| “Final states” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | Off |
| “Signal logging” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | On |
| “Inspect signal logs when simulation is paused/stopped” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | Off |
| “Limit data points to last” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | On |

Mapping Application Requirements to the Data Import/Export Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--------------------------------|-----------------------------------|---------------------|-------------------|------------------------------|------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Decimation” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | 1 |
| “Format” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | Array |
| “Output options” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | Refine output |
| “Refine factor” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | 1 |
| “Output times” | No impact | No impact | No impact | No impact (GRT) Off (ERT) | ' [] ' |

Mapping Application Requirements to the Optimization Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|------------------------------|------------------------------|--------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Block reduction | Off (GRT) No impact (ERT) | Off | On | Off | On |
| Implement logic signals as Boolean data (vs. double) | No impact | No impact | On | On | On |
| Inline parameters | Off (GRT) On (ERT) | On | On | No impact | Off |
| Conditional input branch execution | No impact | On | On | Off | On |
| Signal storage reuse | Off | Off | On | No impact | On |
| Application lifespan (days) | No impact | No impact | Finite value | inf | inf |
| Enable local block outputs | Off | No impact | On | No impact | On |
| Ignore integer downcasts in folded expressions | Off | No impact | No impact | Off | Off |
| Eliminate superfluous local variables (Expression folding) | Off | No impact (GRT) Off (ERT) | On | No impact | On |

Mapping Application Requirements to the Optimization Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|-----------------------------------|---------------------|---|--------------------------|------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Minimize data copies between local and global variables” | Off | Off | On | On | Off |
| Loop unrolling threshold | No impact | No impact | >0 | >1 | 5 |
| Use memcpy for vector assignment | No impact | No impact | On | No impact | On |
| Memcpy threshold (bytes) | No impact | No impact | Accept default or determine target-specific optimal value | No impact | 64 |
| Use memset to initialize floats and doubles to 0.0 | No impact | No impact | On | No impact | On |
| Reuse block outputs | Off | Off | On | No impact | On |
| Inline invariant signals | Off | Off | On | No impact | Off |

Mapping Application Requirements to the Optimization Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|----------------------------|--------------|------------|-----------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Remove code from floating-point to integer conversions that wraps out-of-range values | Off | Off | On | Off (GRT) On (ERT) | Off |
| Remove code from floating-point to integer conversions with saturation that maps NaN to zero | Off | Off | On | Off (GRT) On (ERT) | On |
| “Use bitsets for storing state configuration” | Off | Off | Off | No impact | Off |
| “Use bitsets for storing Boolean data” | Off | Off | Off | No impact | Off |

Mapping Application Requirements to the Diagnostics: Solver Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Algebraic loop” | error | No impact | No impact | error | warning |

Mapping Application Requirements to the Diagnostics: Solver Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Minimize algebraic loop” | No impact | No impact | No impact | error | warning |
| “Block priority violation” | No impact | No impact | No impact | error | warning |
| “Consecutive zero-crossings violation” | No impact | No impact | No impact | warning or error | error |
| “Unspecified inheritability of sample time” | No impact | No impact | No impact | error | warning |
| “Solver data inconsistency” | warning | No impact | none | No impact | warning |
| “Automatic solver parameter selection” | No impact | No impact | No impact | error | warning |

Mapping Application Requirements to the Diagnostics: Sample Time Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Source block specifies -1 sample time” | No impact | No impact | No impact | error | none |
| “Discrete used as continuous” | No impact | No impact | No impact | error | warning |
| “Multitask rate transition” | No impact | No impact | No impact | error | error |

Mapping Application Requirements to the Diagnostics: Sample Time Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Single task rate transition” | No impact | No impact | No impact | none or error | none |
| “Multitask conditionally executed subsystem” | No impact | No impact | No impact | error | error |
| “Tasks with equal priority” | No impact | No impact | No impact | none or error | warning |
| “Enforce sample times specified by Signal Specification blocks” | No impact | No impact | No impact | error | warning |

Mapping Application Requirements to the Diagnostics: Data Validity Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------------|----------------------------|------------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Signal resolution” | No impact | No impact | No impact | Explicit only | Explicit only |
| “Division by singular matrix” | No impact | No impact | No impact | error | none |
| “Underspecified data types” | No impact | No impact | No impact | error | none |
| “Simulation range checking” | warning or error | warning or error | none | error | none |
| “Detect overflow” | No impact | No impact | No impact | error | warning |

Mapping Application Requirements to the Diagnostics: Data Validity Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|-----------------------------------|---------------------|-------------------|--------------------------|------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Inf or NaN block output” | No impact | No impact | No impact | error | none |
| “rt” prefix for identifiers” | No impact | No impact | No impact | error | error |
| “Detect downcast” | No impact | No impact | No impact | error | error |
| “Detect overflow” | No impact | No impact | No impact | error | error |
| “Detect underflow” | No impact | No impact | No impact | error | none |
| “Detect precision loss” | No impact | No impact | No impact | error | error |
| “Detect loss of tunability” | No impact | No impact | No impact | error | none |
| “Detect read before write” | No impact | No impact | No impact | error | Enable all as warnings |
| “Detect write after read” | No impact | No impact | No impact | error | Enable all as warning |
| “Detect write after write” | No impact | No impact | No impact | error | Enable all as errors |
| “Multitask data store” | No impact | No impact | No impact | error | warning |
| “Duplicate data store names” | warning | No impact | none | No impact | none |
| “Check undefined subsystem initial output” | No impact | No impact | No impact | On | On |

Mapping Application Requirements to the Diagnostics: Data Validity Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|--------------------------------------|--------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Check preactivation output of execution context” | No impact | No impact | No impact | On | Off |
| “Check runtime output of execution context” | No impact | No impact | No impact | On | Off |
| Model Verification block enabling | No impact | No impact | No impact | No impact (GRT) Disable all (ERT) | Use local settings |

Mapping Application Requirements to the Diagnostics: Type Conversion Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Unnecessary type conversions” | No impact | No impact | No impact | warning | none |
| “Vector/matrix block input conversion” | No impact | No impact | No impact | error | none |
| “32-bit integer to single precision float conversion” | No impact | No impact | No impact | warning | warning |

Mapping Application Requirements to the Diagnostics: Connectivity Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|-------------------|--------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Signal label mismatch” | No impact | No impact | No impact | error | none |
| “Unconnected block input ports” | No impact | No impact | No impact | error | warning |
| “Unconnected block output ports” | No impact | No impact | No impact | error | warning |
| “Unconnected line” | No impact | No impact | No impact | error | none |
| “Unspecified bus object at root Output block” | No impact | No impact | No impact | error | warning |
| “Element name mismatch” | No impact | No impact | No impact | error | warning |
| “Mux blocks used to create bus signals” | No impact | No impact | No impact | error | warning |
| “Bus signal treated as vector” | No impact | No impact | No impact | error | warning |
| “Invalid function-call connection” | No impact | No impact | No impact | error | error |
| “Context-dependent inputs” | No impact | No impact | No impact | Enable all | Use local settings |

Mapping Application Requirements to the Diagnostics: Compatibility Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|------------------------------|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “S-function upgrades needed” | No impact | No impact | No impact | error | none |

Mapping Application Requirements to the Diagnostics: Model Referencing Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Model block version mismatch” | No impact | No impact | No impact | none | none |
| “Port and parameter mismatch” | No impact | No impact | No impact | error | none |
| “Model configuration mismatch” | No impact | No impact | No impact | warning | none |
| “Invalid root Inport/Outport block connection” | No impact | No impact | No impact | error | none |
| “Unsupported data logging” | No impact | No impact | No impact | error | warning |

Mapping Application Requirements to the Diagnostics: Saving Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Block diagram contains disabled library links” | No impact | No impact | No impact | No impact | warning |
| “Block diagram contains parameterized library links” | No impact | No impact | No impact | No impact | none |

Mapping Application Requirements to the Hardware Implementation Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------|----------------------------|--------------|-----------------|--|-------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Device vendor | No impact | No impact | No impact | No impact | No impact |
| Device type | No impact | No impact | No impact | No impact | No impact |
| Number of bits | No impact | No impact | Target specific | No impact (GRT) Match operation of target compiler and hardware (ERT) | 8, 16, 32, 32, 32 |
| Byte ordering | No impact | No impact | No impact | No impact | Unspecified |

Mapping Application Requirements to the Hardware Implementation Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|------------------------------------|--|-------------------------------|--------------------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Signed integer division rounds to | No impact (GRT) Undefined (ERT) | No impact (GRT) Zero or Floor (ERT) | No impact (GRT) Zero (ERT) | No impact (GRT) Floor (ERT) | Undefined |
| Shift right on a signed integer as arithmetic shift | No impact | No impact | On | No impact | On |
| Emulation hardware (code generation only) | No impact | No impact | No impact | No impact | On |

Mapping Application Requirements to the Model Referencing Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|----------------------------------|-------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Rebuild options” | No impact | No impact | No impact | Never or If any changes detected | If any changes detected |
| “Never rebuild targets diagnostic” | No impact | No impact | No impact | error if targets require rebuild | error |
| “Total number of instances allowed per top model” | No impact | No impact | No impact | No impact | Multiple |

Mapping Application Requirements to the Model Referencing Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|-----------------------------------|---------------------|-------------------|--------------------------|------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Model dependencies” | No impact | No impact | No impact | No impact | '' |
| “Pass scalar root inputs by value for Real-Time Workshop” | No impact | No impact | No impact | Off | Off |
| “Minimize algebraic loop occurrences” | No impact | No impact | No impact | Off | Off |

Mapping Application Requirements to the Simulation Target: General Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|-----------------------------------|---------------------|-------------------|--------------------------|------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Enable debugging/animation” | On | No impact | Off | On | On |
| “Enable overflow detection (with debugging)” | On | No impact | Off | On | On |
| “Ensure memory integrity” | On | On | Off | On | On |
| “Echo expressions without semicolons” | On | No impact | Off | No impact | On |
| “Use BLAS library for faster simulation” | No impact | No impact | On | No impact | On |

Mapping Application Requirements to the Simulation Target: General Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--------------------------------|----------------------------|--------------|------------|-------------------|-------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Ensure responsiveness” | On | On | Off | On | On |
| “Simulation target build mode” | No impact | No impact | No impact | No impact | Incremental build |

Mapping Application Requirements to the Simulation Target: Symbols Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Reserved names” | No impact | No impact | No impact | No impact | { } |

Mapping Application Requirements to the Simulation Target: Custom Code Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Source file” | No impact | No impact | No impact | No impact | '' |
| “Header file” | No impact | No impact | No impact | No impact | '' |
| “Initialize function” | No impact | No impact | No impact | No impact | '' |
| “Terminate function” | No impact | No impact | No impact | No impact | '' |

Mapping Application Requirements to the Simulation Target: Custom Code Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| “Include directories” | No impact | No impact | No impact | No impact | '' |
| “Source files” | No impact | No impact | No impact | No impact | '' |
| “Libraries” | No impact | No impact | No impact | No impact | '' |

Mapping Application Requirements to the Real-Time Workshop: General Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|------------------------------------|-----------------------------------|-----------------------------------|--------------------------------|------------------------------------|-----------------------------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| System target file | No impact | No impact | No impact | No impact (GRT) ERT based (ERT) | grt.tlc |
| Language | No impact | No impact | No impact | No impact | C |
| Compiler optimization level | Optimizations off (faster builds) | Optimizations off (faster builds) | Optimizations on (faster runs) | No impact | Optimizations off (faster builds) |
| Custom compiler optimization flags | Optimizations off (faster builds) | Optimizations off (faster builds) | Optimizations on (faster runs) | No impact | Optimizations off (faster builds) |
| TLC options | No impact | No impact | No impact | No impact | '' |

**Mapping Application Requirements to the Real-Time Workshop: General Pane
(Continued)**

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|--|--|--|--|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Generate makefile | No impact | No impact | No impact | No impact | On |
| Make command | No impact | No impact | No impact | make_rtw | make_rtw |
| Template makefile | No impact | No impact | No impact | No impact | grt_default_tmf |
| “Select objective” on page 6-25 | Debugging | Not applicable for GRT-based targets | Not applicable for GRT-based targets | Not applicable for GRT-based targets | Unspecified |
| “Check model before generating code” on page 6-32 | On (proceed with warnings) or On (stop for warnings) | On (proceed with warnings) or On (stop for warnings) | On (proceed with warnings) or On (stop for warnings) | On (proceed with warnings) or On (stop for warnings) | Off |
| Generate code only | Off | No impact | No impact | No impact | Off |

Mapping Application Requirements to the Real-Time Workshop: Report Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|----------------------------|--------------|------------|--------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precautions | |
| “Create code generation report” on page 6-40 | On | On | No impact | On | Off |
| “Launch report automatically” on page 6-43 | On | On | No impact | No impact | Off |

Mapping Application Requirements to the Real-Time Workshop: Comments Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Include comments | On | On | No impact | On | On |
| Simulink block / Stateflow object comments | On | On | No impact | On | On |
| Show eliminated blocks | On | On | No impact | On | Off |
| Verbose comments for Simulink Global storage class | On | On | No impact | On | Off |

Mapping Application Requirements to the Real-Time Workshop: Symbols Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Maximum identifier length | Any valid value | >30 | No impact | >30 | 31 |
| Use the same reserved names as Simulation Target | No impact | No impact | No impact | No impact | Off |
| Reserved names | No impact | No impact | No impact | No impact | { } |

Mapping Application Requirements to the Real-Time Workshop: Custom Code Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|--|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Use the same custom code settings as Simulation Target | No impact | No impact | No impact | No impact | Off |
| Source file | No impact | No impact | No impact | No impact | '' |
| Header file | No impact | No impact | No impact | No impact | '' |
| Initialize function | No impact | No impact | No impact | No impact | '' |
| Terminate function | No impact | No impact | No impact | No impact | '' |
| Include directories | No impact | No impact | No impact | No impact | '' |

Mapping Application Requirements to the Real-Time Workshop: Custom Code Pane (Continued)

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|-------------------------|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Source files | No impact | No impact | No impact | No impact | '' |
| Libraries | No impact | No impact | No impact | No impact | '' |

Mapping Application Requirements to the Real-Time Workshop: Debug Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|----------------------------|--------------|------------|-------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Verbose build | On | No impact | No impact | On | On |
| Retain .rtw file | On | No impact | No impact | No impact | Off |
| “Profile TLC” on page 6-137 | On | No impact | No impact | No impact | Off |
| Start TLC debugger when generating code | On | No impact | No impact | No impact | Off |
| Start TLC coverage when generating code | On | No impact | No impact | No impact | Off |
| Enable TLC assertion | On | No impact | No impact | On | Off |

Mapping Application Requirements to the Real-Time Workshop: Interface Pane

| Configuration Parameter | Settings for Building Code | | | | Factory Default |
|---|--|--|-----------------|-------------------------------|-----------------|
| | Debugging | Traceability | Efficiency | Safety Precaution | |
| Target function library | No impact | No impact | Any valid value | No impact | C89/C90 (ANSI) |
| Utility function generation | Shared location (GRT) No impact (ERT) | Shared location (GRT) No impact (ERT) | Shared location | No impact | Auto |
| MAT-file variable name modifier | No impact | No impact | No impact | No impact | rt_ |
| Interface | No impact | No impact | No impact | No impact (GRT) None (ERT) | None |
| Signals in C API | No impact | No impact | No impact | No impact | On |
| Parameters in C API | No impact | No impact | No impact | No impact | On |
| Transport layer | No impact | No impact | No impact | No impact | tcpip |
| MEX-file arguments | No impact | No impact | No impact | No impact | ' ' |
| Static memory allocation | No impact | No impact | No impact | No impact | Off |
| “Static memory buffer size” on page 6-259 | No impact | No impact | No impact | No impact | 1000000 |

Parameter Command-Line Information Summary

The following table lists Real-Time Workshop parameters that you can use to tune model and target configurations. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to Configuration Parameter dialog box equivalents. For descriptions of the panes and options in that dialog box, see Configuration Parameters in the Real-Time Workshop documentation.

Use the `get_param` and `set_param` commands to retrieve and set the values of the parameters on the MATLAB command line or programmatically in scripts.

The Configuration Wizard in the Real-Time Workshop Embedded Coder product provides buttons and scripts for customizing code generation. See “Using Configuration Wizard Blocks” in the Real-Time Workshop Embedded Coder documentation for information on using Configuration Wizard features.

For information about Simulink parameters, see “Configuration Parameters Dialog Box” in the Simulink documentation. For information on using `get_param` and `set_param` to tune the parameters for various model configurations, see “Parameter Tuning by Using MATLAB Commands”.

For parameters that are specific to the ERT target, or targets based on the ERT target, see “Parameter Command-Line Information Summary” in the Real-Time Workshop Embedded Coder documentation.

Note Parameters that are specific to Stateflow or Simulink® Fixed Point™ products are marked with (Stateflow) and (Simulink Fixed Point), respectively.

The default setting for a parameter might vary for different targets.

Command-Line Information: Optimization Pane

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|--|---|
| BooleanDataType off, on | Optimization > Implement logic signals as Boolean data (vs. double) | Control the output data type of blocks that generate logic signals. |
| BufferReuse off, on | Optimization > Reuse block outputs | Reuse local (function) variables for block outputs wherever possible. Selecting this option trades code traceability for code efficiency. |
| DataBitsets (Stateflow) off , on | Optimization > Use bitsets for storing Boolean data | Use bit sets for storing Boolean data. |
| EfficientFloat2IntCast off , on | Optimization > Remove code from floating-point to integer conversions that wrap out-of-range values | Remove wrapping code that handles out-of-range floating-point to integer conversion results. |
| EfficientMapNaN2IntZero off, on | Optimization > Remove code from floating-point to integer conversions with saturation that maps NaN to zero | Remove code that handles floating-point to integer conversion results for NaN values. |
| EnableMemcpy off, on | Optimization > Use memcpy for vector assignment | Optimize code generated for vector assignment by replacing for loops with memcpy function calls. |

Command-Line Information: Optimization Pane (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|--|--|
| EnforceIntegerDowncast off, on | Optimization > Ignore integer downcasts in folded expressions | Remove casts of intermediate variables. When you set this option to on, 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. Note EnforceIntegerDowncast will be removed in a future release. |
| EnhancedBackFolding off , on | Optimization > Minimize data copies between local and global variables | Reuse existing global variables to store temporary results. |
| ExpressionFolding off, on | Optimization > Eliminate superfluous local variables (Expression folding) > Interface | Collapse block computations into single expressions wherever possible. This improves code readability and efficiency. |
| InitFltsAndDblsToZero off , on | Optimization > Use memset to initialize floats and doubles to 0.0 | Optimize initialization of storage for float and double values. Set this option if the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern 0. |
| InlineInvariantSignals off, on | Optimization > Inline invariant signals | Precompute and inline the values of invariant signals in the generated code. |

Command-Line Information: Optimization Pane (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---|--|--|
| LifeSpan <i>string</i> | 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. |
| MemcpyThreshold int - 64 | Optimization > Memcpy threshold (bytes) | Specify the minimum array size in bytes for which memcpy function calls should replace for loops in the generated code for vector assignments. |
| NoFixptDivByZeroProtection (Simulink Fixed Point) 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. |
| RollThreshold int - 5 | Optimization > Loop unrolling threshold | Specify the minimum signal width for which a for loop is to be generated. |
| StateBitsets (Stateflow) off, on | Optimization > Use bitsets for storing state configuration | Use bit sets for storing state configuration. |
| UseIntDivNetSlope (Simulink Fixed Point) off, on | Optimization > Use integer division to handle net slopes that are reciprocals of integers | Perform net slope correction using integer division when simplicity and accuracy conditions are met. |

Command-Line Information: Real-Time Workshop Pane: General Tab

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|--|---|
| CheckMdlBeforeBuild <i>string</i> - off , warning, error | Real-Time Workshop > General > Check model before generating code | Specify whether to run Code Generation Advisor checks before generating code. |
| GenCodeOnly <i>string</i> - off , on | Real-Time Workshop > General > Generate code only | Generate source code, but do not execute the makefile to build an executable. |
| GenerateMakefile <i>string</i> - off , on | Real-Time Workshop > General > Generate makefile | Specify whether to generate a makefile during the build process for a model. |
| MakeCommand <i>string</i> - 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. |
| ObjectivePriorities (GRT) <i>string</i> - { }, {'Debugging'} | Real-Time Workshop > General > Select objective | Specify the code generation objectives to use with the Code Generation Advisor. |
| ObjectivePriorities (ERT) <i>string</i> - { }, {'Efficiency'}, {'Traceability'}, {'Safety precaution'}, {'Debugging'} | Real-Time Workshop > General > Set objectives | Specify and prioritize the code generation objectives to use with the Code Generation Advisor. |

Command-Line Information: Real-Time Workshop Pane: General Tab (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|--|---|
| RTWCompilerOptimization string - Off , On, Custom | Real-Time Workshop > General > Compiler optimization level | Use this parameter to trade off compilation time against run time for your model code without having to supply compiler-specific flags to other levels of the Real-Time Workshop build process. Off - Turn compiler optimizations off for faster builds On - Turn compiler optimizations on for faster code execution Custom - Specify custom compiler optimization flags via the RTWCustomCompilerOptimizations parameter |
| RTWCustomCompiler Optimizations string - , unquoted string of compiler optimization flags | Real-Time Workshop > General > Custom compiler optimization flags | If you specified Custom to the RTWCompilerOptimization parameter, use this parameter to specify custom compiler optimization flags, for example, -O2. |
| SaveLog off , on | Real-Time Workshop > General > Save build log | Save build log. |
| SystemTargetFile string - grt.tlc | Real-Time Workshop > General > System target file | Specify a system target file. |

Command-Line Information: Real-Time Workshop Pane: General Tab (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|---|---|
| TargetLang string - C , C++, C++ (Encapsulated) (ERT) | Real-Time Workshop > General > Language | Specify whether to generate C code, C++ compatible code, or C++ encapsulated code. The C++ (Encapsulated) value appears only when you select an ERT system target file for the model. Using C++ (Encapsulated) to generate code requires a Real-Time Workshop Embedded Coder license. |
| TemplateMakefile string - grt_default_tmf | Real-Time Workshop > General > Template makefile | Specify the current template makefile for building a Real-Time Workshop target. |
| TLCOptions string - | Real-Time Workshop > General > TLC options | Specify additional TLC command line options. |

Command-Line Information: Real-Time Workshop Pane: Report Tab

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|--|--|
| GenerateReport string - off , on | Real-Time Workshop > Report > Create code generation report | Document the generated C or C++ code in an HTML report. |
| LaunchReport string - off , on | Real-Time Workshop > Report > Launch report automatically | Display the HTML report after code generation completes. |

Command-Line Information: Real-Time Workshop Pane: Comments Tab

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---|--|--|
| ForceParamTrailComments <i>string</i> - off , on | Real-Time Workshop > Comments > Verbose comments for SimulinkGlobal storage class | Specify that comments be included in the generated file. To reduce file size, the model parameters data structure is not commented when there are more than 1000 parameters. |
| GenerateComments <i>string</i> - off , on | Real-Time Workshop > Comments > Include comments | Include comments in generated code. |
| ShowEliminatedStatement <i>string</i> - off , on | Real-Time Workshop > Comments > Show eliminated blocks | Show statements for eliminated blocks as comments in the generated code. |
| SimulinkBlockComments <i>string</i> - off , on | Real-Time Workshop > Comments > Simulink block / Stateflow object comments | Insert Simulink block and Stateflow object names as comments above the generated code for each block. |

Command-Line Information: Real-Time Workshop Pane: Symbols Tab

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---------------------------------------|---|---|
| MaxIdLength <i>int</i> - 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. |

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|--|--|
| ReservedNameArray <i>string array - {}</i> | Real-Time Workshop > Symbols > Reserved names | Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code to avoid name conflicts. |
| UseSimReservedNames <i>string - off, on</i> | Real-Time Workshop > Symbols > Use the same reserved names as Simulation Target | Specify whether to use the same reserved names as those specified in the Simulation Target > Symbols pane. |

Command-Line Information: Real-Time Workshop Pane: Custom Code Tab

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|-------------------------------------|---|---|
| CustomHeaderCode <i>string -</i> | Real-Time Workshop > Custom Code > Header file | Specify the code to appear at the top of the generated <i>model.h</i> header file. |
| CustomInclude <i>string -</i> | Real-Time Workshop > Custom Code > Include directories | Specify a space-separated list of include directories to add to the include path when compiling the generated code. |

Command-Line Information: Real-Time Workshop Pane: Custom Code Tab (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|--|--|
| | | <hr/> <p>Note If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example,</p> <pre>'C:\Project "C:\Custom Files"'</pre> <hr/> |
| <p>CustomInitializer <i>string</i> -</p> | <p>Real-Time Workshop > Custom Code</p> | <p>Specify the code to appear in the generated model initialize function.</p> |
| <p>CustomLibrary <i>string</i> -</p> | <p>Real-Time Workshop > Custom Code > Initialize function Libraries</p> | <p>Specify a space-separated list of static library files to link with the generated code.</p> <hr/> <p>Note If your list includes any Windows file name strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example,</p> <pre>'mathutil.lib "op utils.lib"'</pre> <hr/> |

Command-Line Information: Real-Time Workshop Pane: Custom Code Tab (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---|--|---|
| CustomSource <i>string</i> - | Real-Time Workshop > Custom Code > Source files | Specify a space-separated list of source files to compile and link with the generated code. Note If your list includes any Windows file name strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, <code>'algs.c "div alg.c"'</code> |
| CustomSourceCode <i>string</i> - | Real-Time Workshop > Custom Code > Source file | Specify code to appear at the top of the generated <i>model.c</i> source file. |
| CustomTerminator <i>string</i> - | Real-Time Workshop > Custom Code > Terminate function | Specify code to appear in the model generated terminate function. |
| RTWUseSimCustomCode <i>string</i> - off , on | Real-Time Workshop > Custom Code > Use the same custom code settings as Simulation Target | Specify whether to use the same custom code settings as those in the Simulation Target > Custom Code pane. |

Command-Line Information: Real-Time Workshop Pane: Debug Tab

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---|---|--|
| ProfileTLC string - off , on | Real-Time Workshop > Debug > Profile TLC | Profile the execution time of each TLC file used to generate code for this model in HTML format. |
| RTWVerbose string - off , on | Real-Time Workshop > Debug > Verbose build | Display messages indicating code generation stages and compiler output. |
| RetainRTWFile string - off , on | Real-Time Workshop > Debug > Retain .rtw file | Retain the <i>model.rtw</i> file in the current build directory. |
| TLCAssert string - off , on | Real-Time Workshop > Debug > Enable TLC assertion | Produce a TLC stack trace when the argument to the <code>assert</code> directives evaluates to false. |
| TLCCoverage string - off , on | Real-Time Workshop > Debug > Start TLC coverage when generating code | Generate <code>.log</code> files containing the number of times each line of TLC code is executed during code generation. |
| TLCDebug string - off , on | Real-Time Workshop > Debug > Start TLC debugger when generating code | Start the TLC debugger during code generation at the beginning of the TLC program. TLC breakpoint statements automatically invoke the TLC debugger regardless of this setting. |

Command-Line Information: Real-Time Workshop Pane: Interface Tab

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|---|--|
| ExtMode off , on | Real-Time Workshop > Interface > Interface | Specify the data interface to be generated with the code. |
| ExtModeMexArgs <i>string</i> () | Real-Time Workshop > Interface > Interface > External > MEX-file arguments | Specify arguments that are passed to an external mode interface MEX-file for communicating with executing targets. |
| ExtModeStaticAlloc off , on | Real-Time Workshop > Interface > Static memory allocation | Use a static memory buffer for external mode instead of allocating dynamic memory (calls to malloc). |
| ExtModeStaticAllocSize <i>integer</i> (1000000) | Real-Time Workshop > Interface > Static memory buffer size | Specify the size in bytes of the external mode static memory buffer. |
| ExtModeTransport int - 0 for TCP/IP, 1 for 32-bit Windows serial | Real-Time Workshop > Interface > Interface > External > Transport layer | Specify transport protocols for external mode communications. |
| GenerateASAP2 off , on | Real-Time Workshop > Interface > Interface | Specify the data interface to be generated with the code. |

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---|--|--|
| <p>GenFloatMathFcnCalls string - ANSI_C, C99 (ISO), GNU99 (GNU)</p> <p>(For ERT-based models, additional target-specific values may be available; see the Target function library drop-down list in the Configuration Parameters dialog box.)</p> | <p>Real-Time Workshop > Interface > Target function library</p> | <p>Specify a target-specific math library for your model. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur.</p> <p>ANSI_C - ISO/IEC 9899:1990 C standard math library for floating-point functions C99 (ISO) - ISO/IEC 9899:1999 C standard math library GNU99 (GNU) - GNU gcc math library, which provides C99 extensions as defined by compiler option <code>-std=gnu99</code></p> |
| <p>LogVarNameModifier string - none, <code>rt_</code>, <code>_rt</code></p> | <p>Real-Time Workshop > Interface > MAT-file variable name modifier</p> | <p>Augment the MAT-file variable name.</p> |
| <p>MatFileLogging (ERT) string - off, <code>on</code></p> | <p>Real-Time Workshop > Interface > MAT-file logging</p> | <p>Generate code that logs data to a MAT-file.</p> |
| <p>RTWCAPIParams string - off, <code>on</code></p> | <p>Real-Time Workshop > Interface > Parameters in C API</p> | <p>Generate parameter tuning structures in C API.</p> |

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---|--|--|
| RTWCAPISignals string - off , on | Real-Time Workshop > Interface > Signals in C API | Generate signal structure in C API. |
| UtilityFuncGeneration string - Auto , Shared location | Real-Time Workshop > Interface > Utility function generation | Specify where utility functions are to be generated. |

Command-Line Information: Not in GUI

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|----------------------------------|--|---|
| CodeGenDirectory | Not available | For MathWorks™ use only. |
| Comment | Not available | For MathWorks use only. |
| CompOptLevelCompliant off, on | Not available | Set in SelectCallback for a target to indicate whether the target supports the ability to use the Compiler optimization level parameter on the Real-Time Workshop pane to control the compiler optimization level for building generated code. Default is off for custom targets and on for targets provided with the Real-Time Workshop and Real-Time Workshop Embedded Coder products. |
| ConfigAtBuild | Not available | For MathWorks use only. |
| ConfigurationMode | Not available | For MathWorks use only. |

Command-Line Information: Not in GUI (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|---|--|
| ConfigurationScript | Not available | For MathWorks use only. |
| ERTCustomFileBanners | Not available | For MathWorks use only. |
| EvaledLifeSpan | Not available | For MathWorks use only. |
| ExtModeMexFile | Not available | For MathWorks use only. |
| ExtModeTesting | Not available | For MathWorks use only. |
| FoldNonRolledExpr | Not available | For MathWorks use only. |
| GenerateFullHeader | Not available | For MathWorks use only. |
| IncAutoGenComments | Not available | For MathWorks use only. |
| IncludeRegionsInRTWFile BlockHierarchyMap | Not available | For MathWorks use only. |
| IncludeRootSignalInRTWFile | Not available | For MathWorks use only. |
| IncludeVirtualBlocksInRTW FileBlockHierarchyMap | Not available | For MathWorks use only. |
| IsERTTarget | Not available | For MathWorks use only. |
| IsPILTarget | Not available | For MathWorks use only. |
| ModelReferenceCompliant string - off, on | Not available | Set in <code>SelectCallback</code> for a target to indicate whether the target supports model reference. |
| ParamNamingFcn | Not available | For MathWorks use only. |
| PostCodeGenCommand string - | Not available | Add the specified post code generation command to the model build process. |
| PreserveName | Not available | For MathWorks use only. |
| PreserveNameWithParent | Not available | For MathWorks use only. |
| ProcessScript | Not available | For MathWorks use only. |

Command-Line Information: Not in GUI (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|--|---|---|
| ProcessScriptMode | Not available | For MathWorks use only. |
| RTWCAPIStates | Not available | For MathWorks use only. |
| SignalNamingFcn | Not available | For MathWorks use only. |
| SystemCodeInlineAuto | Not available | For MathWorks use only. |
| TargetFcnLib | Not available | For MathWorks use only. |
| TargetLibSuffix <i>string</i> - | Not available | Control the suffix used for naming a target's dependent libraries (for example, <code>_target.lib</code> or <code>_target.a</code>). If specified, the string must include a period (.). (For generated model reference libraries, the library suffix defaults to <code>_rtwlib.lib</code> on Windows systems and <code>_rtwlib.a</code> on UNIX systems.) |
| TargetPreCompLibLocation <i>string</i> - | Not available | Control the location of precompiled libraries. If you do not set this parameter, the code generator uses the location specified in <code>rtwmakecfg.m</code> . |
| TargetPreprocMaxBitsSint <i>int</i> - 32 | Not available | Specify the maximum number of bits that the target C preprocessor can use for signed integer math. |

Command-Line Information: Not in GUI (Continued)

| Parameter and Values | Configuration Parameters Dialog Box Equivalent | Description |
|---|---|--|
| TargetPreprocMaxBitsUint int - 32 | Not available | Specify the maximum number of bits that the target C preprocessor can use for unsigned integer math. |
| TargetTypeEmulationWarn SuppressLevel SuppressLevel int - 0 | Not available | When greater than or equal to 2, suppress warning messages that the Real-Time Workshop software displays when emulating integer sizes in rapid prototyping environments. |

Configuration Parameters for Embedded MATLAB Coder

- “Real-Time Workshop Dialog Box for Embedded MATLAB Coder” on page 7-2
- “Automatic C MEX Generation Dialog Box for Embedded MATLAB Coder” on page 7-25
- “Hardware Implementation Dialog Box for Embedded MATLAB Coder” on page 7-40
- “Compiler Options Dialog Box” on page 7-49

Real-Time Workshop Dialog Box for Embedded MATLAB Coder

In this section...

“Real-Time Workshop Dialog Box Overview” on page 7-2

“General Tab” on page 7-3

“Report Tab” on page 7-6

“Symbols Tab” on page 7-7

“Custom Code Tab” on page 7-13

“Debug Tab” on page 7-16

“Interface Tab” on page 7-18

“Advanced Tab” on page 7-23

“Generate code only” on page 7-24

Real-Time Workshop Dialog Box Overview

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

Displaying the Dialog Box

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

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

```
codegen_cfg=emlcoder.RTWConfig('system_target');
```

Set `system_target` to:

- **grt** for GRT-based targets (default)
- **ert** for ERT-based targets

Note ERT-based targets require a Real-Time Workshop Embedded Coder license when generating code.

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

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

```
open codegen_cfg;
```

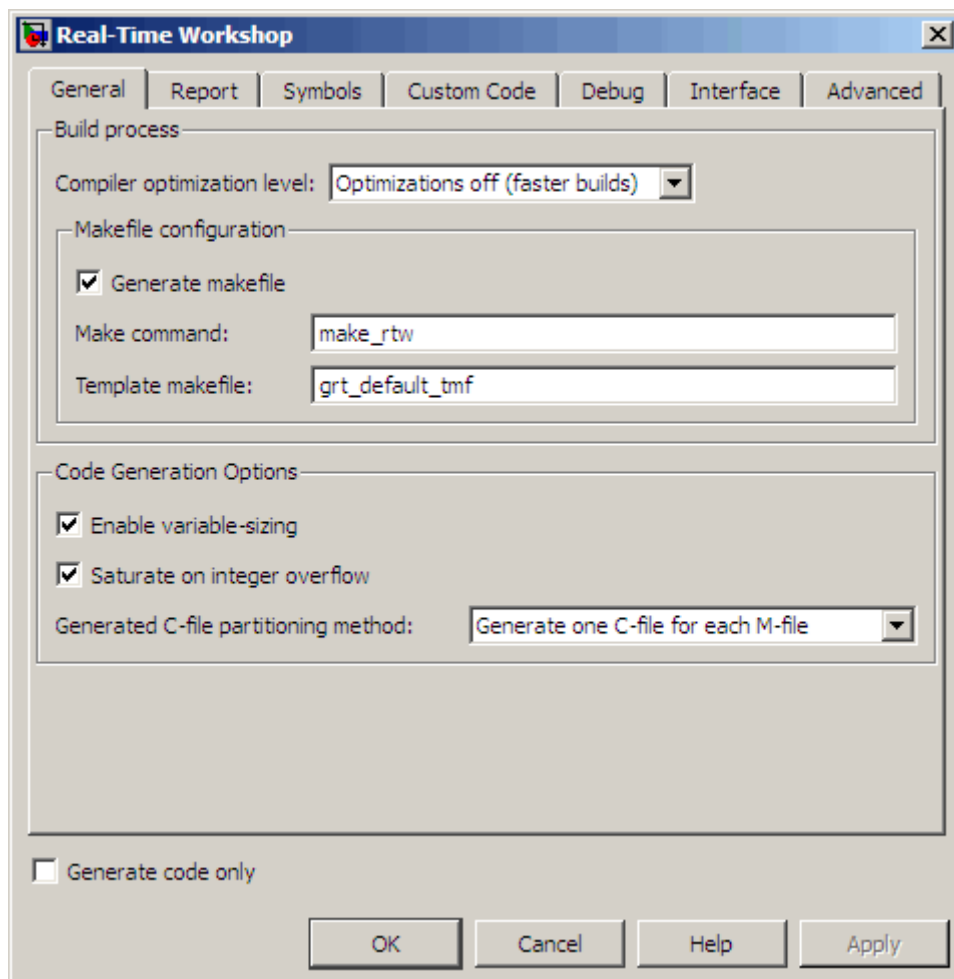
The dialog box displays on your desktop.

See Also

“Configuring Your Environment for Code Generation”

General Tab

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



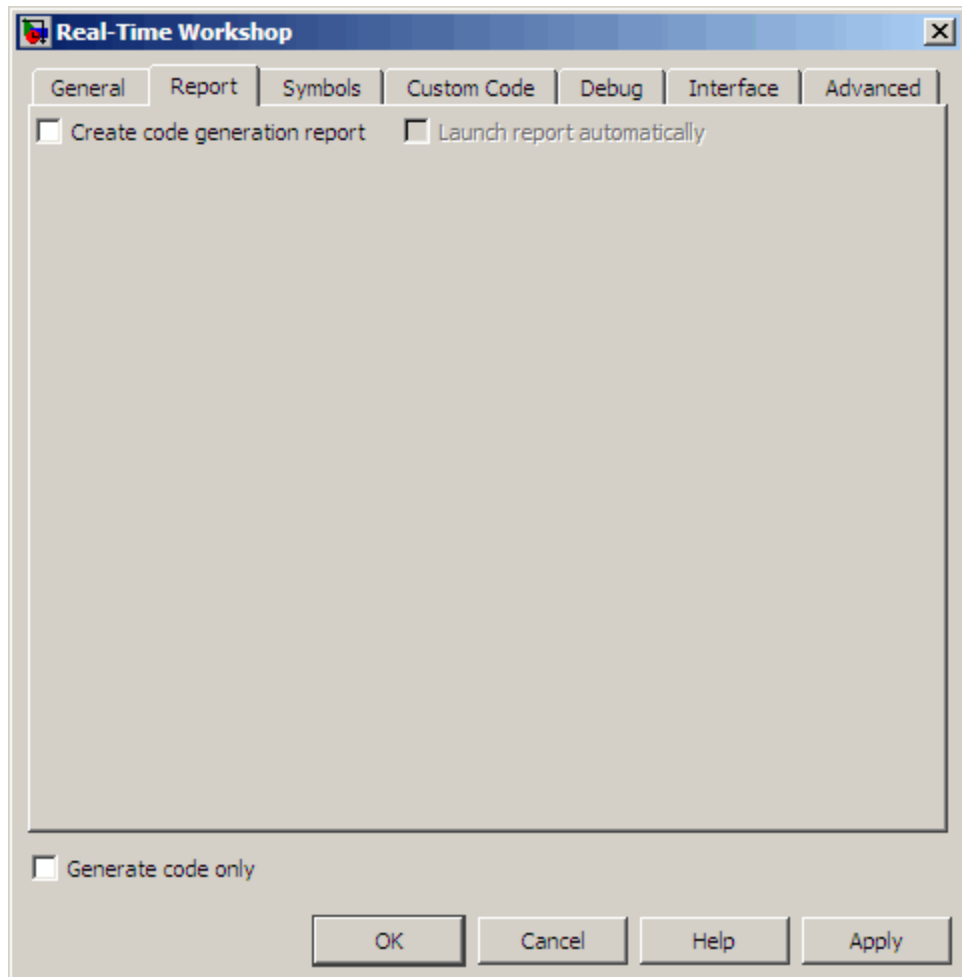
Parameters

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

| General Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---|--|--|
| “Compiler optimization level” on page 6-10 | RTWCompilerOptimization <i>string</i> , Off , 'On', 'Custom' | Specify level of compiler optimization for generating code. Turning optimizations off shortens compile time; turning optimizations on minimizes run time. |
| “Custom compiler optimization flags” on page 6-12 | RTWCustomCompilerOptimizations <i>string</i> , | Specify compiler optimization flags to apply to the generated code. <hr/> Note Requires that you select Custom for Compiler optimization level <hr/> |
| “Generate makefile” on page 6-15 | GenerateMakefile true , false | Specify whether to generate a makefile during the build process. |
| “Make command” on page 6-17 | MakeCommand <i>string</i> , make_rtw | Specify a make command (if Generate makefile is selected). |
| “Template makefile” on page 6-19 | TemplateMakeFile <i>string</i> , grt_default_tmf | Specify a template makefile (if Generate makefile is selected). |
| Enable variable-sizing | EnableVariableSizing true , false | Enable support for variable-sized arrays. |
| Saturate on integer overflow | SaturateOnIntegerOverflow true , false | Add checks in the generated code to detect integer overflow or underflow. |
| Generated C-file partitioning method | FilePartitionMethod <i>string</i> , MapMFileToCFile , SingleFile | Specify whether to generate one C-file for each M-file or generate all C functions into a single file. |

Report Tab

Controls the report that is created for embeddable C code generation using Embedded MATLAB Coder.



Parameters

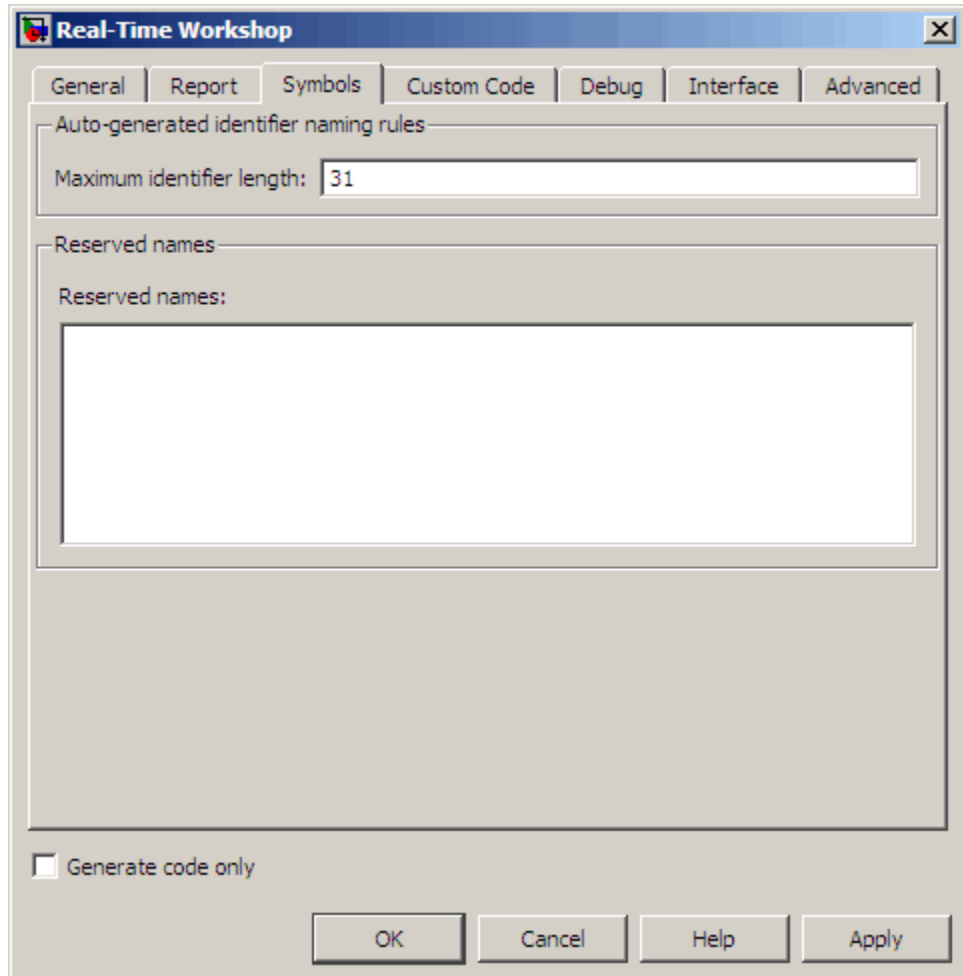
The following table describes the report parameters for the Embedded MATLAB Coder **Real-Time Workshop** dialog box:

| Report Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|--|--|---|
| “Create code generation report” on page 6-40 | GenerateReport true, false | Document generated code in an HTML report. |
| “Launch report automatically” on page 6-43 | LaunchReport true, false | Specify whether to automatically display HTML reports after code generation completes. <hr/> Note Requires that you select Create code generation report <hr/> |

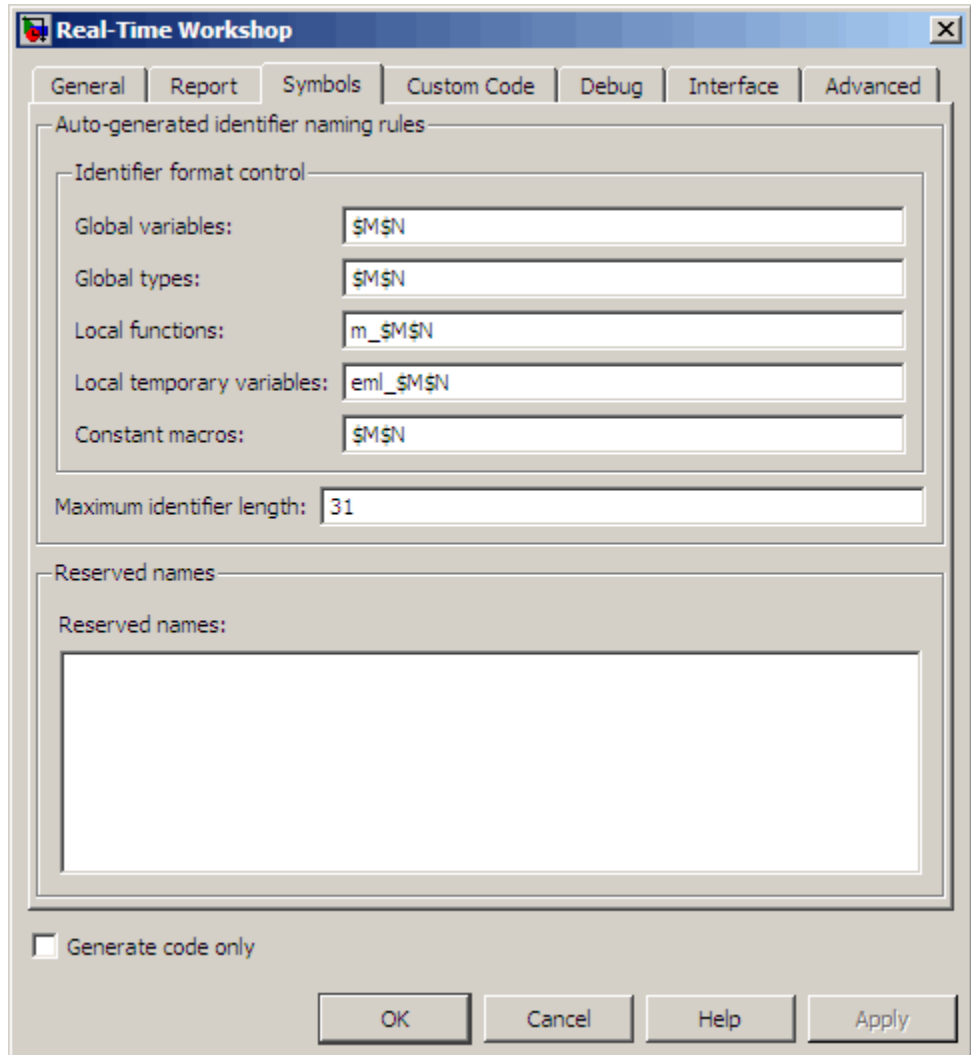
Symbols Tab

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

The Symbols tab appears as follows for GRT-based targets:



The Symbols tab appears as follows for ERT-based targets:



Parameters

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

| Symbols Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|--------------------------|---|---|
| Global variables | CustomSymbolStrGlobalVar <i>string</i> , \$M\$N For more details, see “Settings” on page 7-12 | Customize generated global variable identifiers. Dependencies: <ul style="list-style-type: none"> • This parameter only appears for ERT-based targets. • This parameter requires a Real-Time Workshop Embedded Coder license when generating code. |
| Global types | CustomSymbolStrType <i>string</i> , \$M\$N For more details, see “Settings” on page 7-12 | Customize generated global type identifiers. Dependencies: <ul style="list-style-type: none"> • This parameter only appears for ERT-based targets. • This parameter requires a Real-Time Workshop Embedded Coder license when generating code. |

| Symbols Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---------------------------|--|--|
| Local functions | CustomSymbolStrFcn <i>string</i> , m_ <i>\$M\$N</i> For more details, see “Settings” on page 7-12 | Customize generated local function identifiers. Dependencies: <ul style="list-style-type: none"> • This parameter only appears for ERT-based targets. • This parameter requires a Real-Time Workshop Embedded Coder license when generating code. |
| Local temporary variables | CustomSymbolStrTmpVar <i>string</i> , eml_ <i>\$M\$N</i> For more details, see “Settings” on page 7-12 | Customize generated local temporary variable identifiers. Dependencies: <ul style="list-style-type: none"> • This parameter only appears for ERT-based targets. • This parameter requires a Real-Time Workshop Embedded Coder license when generating code. |

| Symbols Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---|--|--|
| Constant macros | CustomSymbolStrMacro <i>string</i> , \$M\$N For more details, see “Settings” on page 7-12 | Customize generated constant macro identifiers. Dependencies: <ul style="list-style-type: none"> • This parameter only appears for ERT-based targets. • This parameter requires a Real-Time Workshop Embedded Coder license when generating code. |
| “Maximum identifier length” on page 6-102 | MaxIdLength <i>integer</i> , 31 | Specify maximum number of characters in generated function, type definition, and variable names. Minimum is 31. |
| “Reserved names” on page 6-114 | ReservedNameArray <i>string</i> , | Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code. |

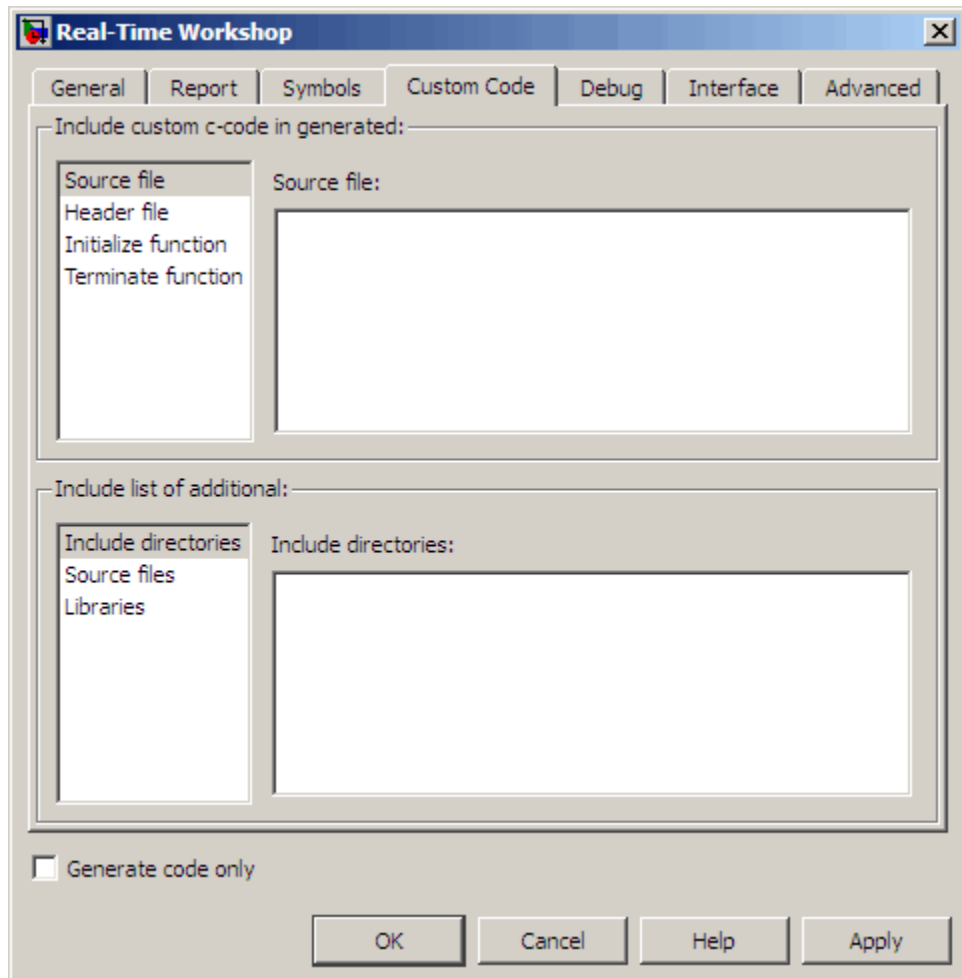
Settings. Enter a macro string that specifies whether, and in what order, certain substrings are to be included in the generated identifier. The macro string can include valid C-identifier characters and a combination of the following format tokens:

| Token | Description |
|--------------|--|
| \$M | Insert name mangling string to avoid naming collisions. Required. |

| Token | Description |
|-------|--|
| \$N | Insert name of parameter (global variable, global type, local function, local temporary variable or constant macro) for which identifier is being generated. Recommended to ensure readability of generated code. |
| \$R | Insert root project name into identifier, replacing any unsupported characters with the underscore (_) character. |

Custom Code Tab

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



Configuration

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

Parameters

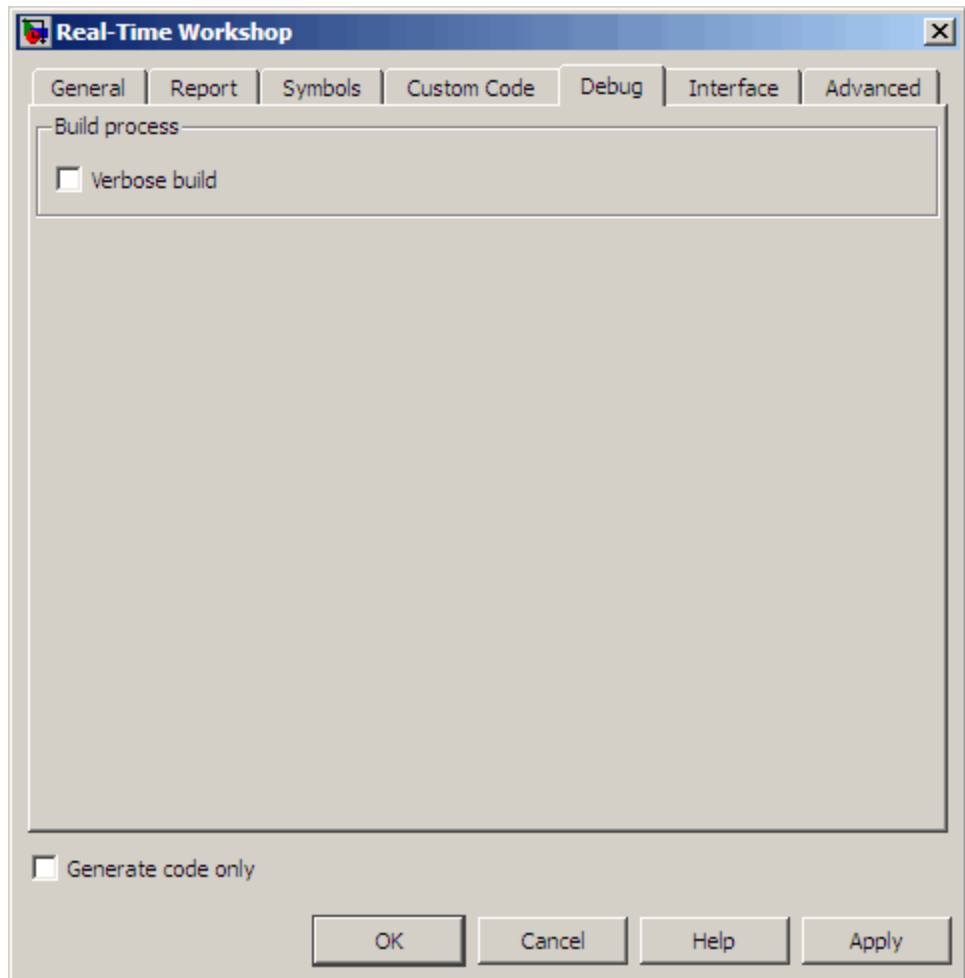
The following table describes the custom code parameters for the Embedded MATLAB Coder **Real-Time Workshop** dialog box:

| Custom Code Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|-------------------------------------|---|--|
| “Source file” on page 6-122 | CustomSourceCode <i>string</i> , | Specify code appearing near the top of the generated .c or .cpp file, outside of any function. |
| “Header file” on page 6-123 | CustomHeaderCode <i>string</i> , | Specify code appearing near the top of the generated .h file. |
| “Initialize function” on page 6-124 | CustomInitializer <i>string</i> , | Specify code appearing in the initialize function of the generated .c or .cpp file. |
| “Terminate function” on page 6-125 | CustomTerminator <i>string</i> , | Specify code appearing in the terminate function of the generated .c or .cpp file. |
| “Include directories” on page 6-126 | CustomInclude <i>string</i> , | Specify a space-separated list of include directories to add to the include path when compiling the generated code. Note If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, <code>'C:\Project "C:\Custom Files"'</code> |

| Custom Code Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|------------------------------|---|--|
| “Source files” on page 6-128 | CustomSource <i>string</i> , | <p>Specify a space-separated list of source files to compile and link with the generated code.</p> <hr/> <p>Note If your list includes any Windows file name strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example,</p> <pre style="text-align: center;">'algs.c "div alg.c"'</pre> <hr/> |
| “Libraries” on page 6-130 | CustomLibrary <i>string</i> , | <p>Specify a space-separated list of static library files to link with the generated code.</p> <hr/> <p>Note If your list includes any Windows file name strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example,</p> <pre style="text-align: center;">'mathutil.lib "op utils.lib"'</pre> <hr/> |

Debug Tab

Specifies parameters for debugging the Embedded MATLAB Coder build process.



Parameters

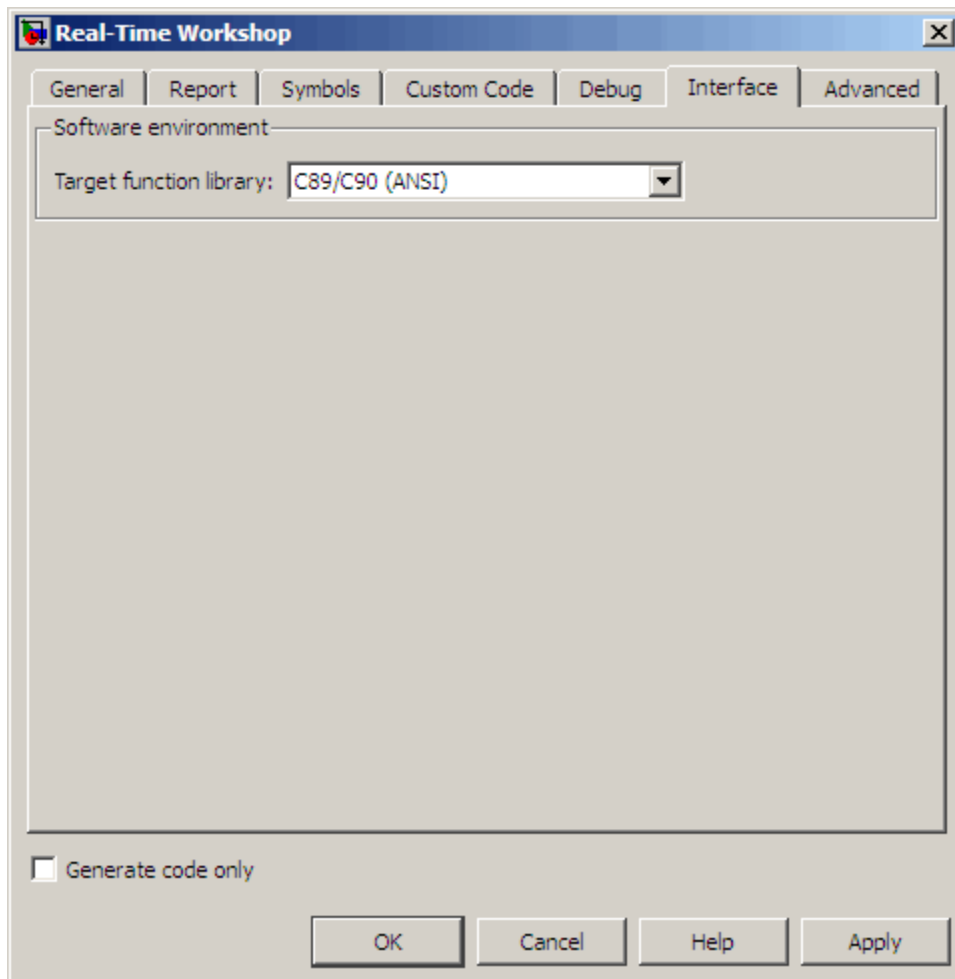
The following table describes the debug parameters for the Embedded MATLAB Coder **Real-Time Workshop** dialog box:

| Debug Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|-------------------------------|--|-----------------------------------|
| “Verbose build” on page 6-135 | RTWVerbose true, false | Display code generation progress. |

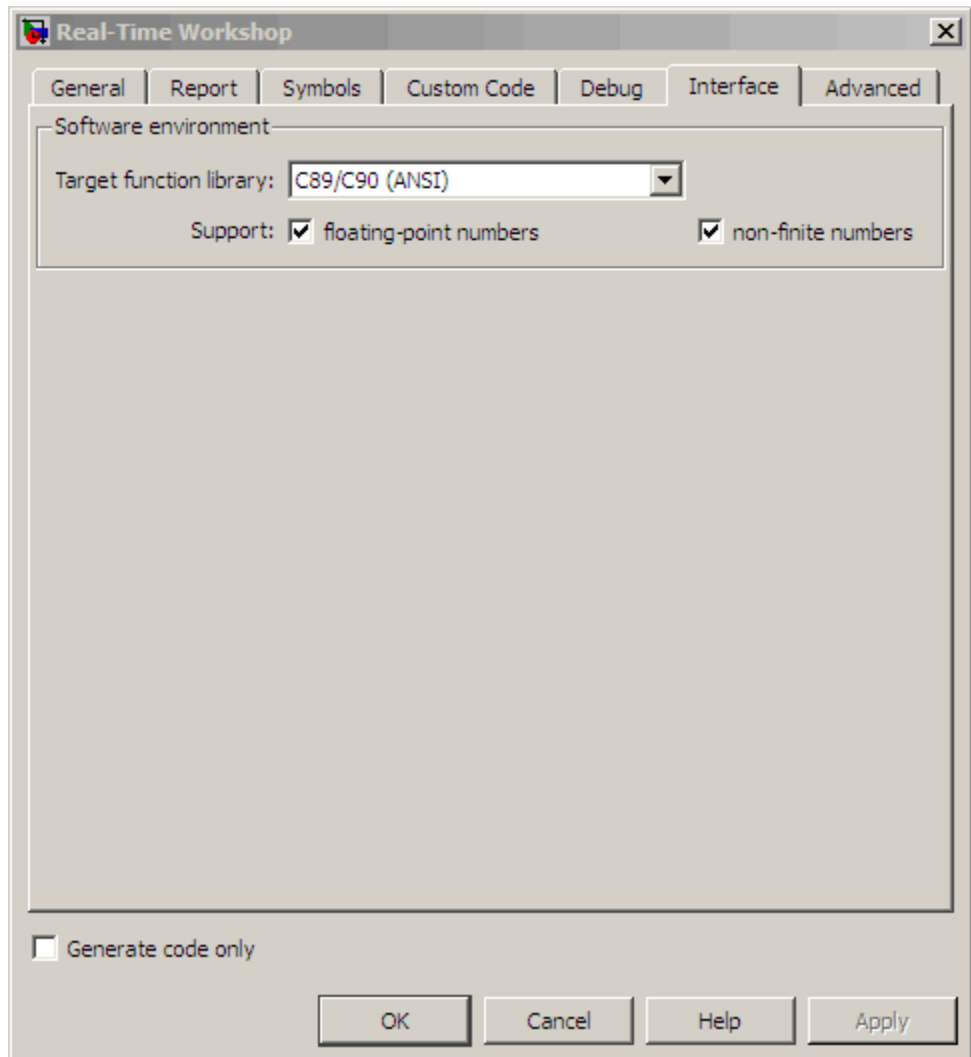
Interface Tab

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

The Interface tab appears as follows for GRT-based targets:



The Interface tab appears as follows for ERT-based targets:



Parameters

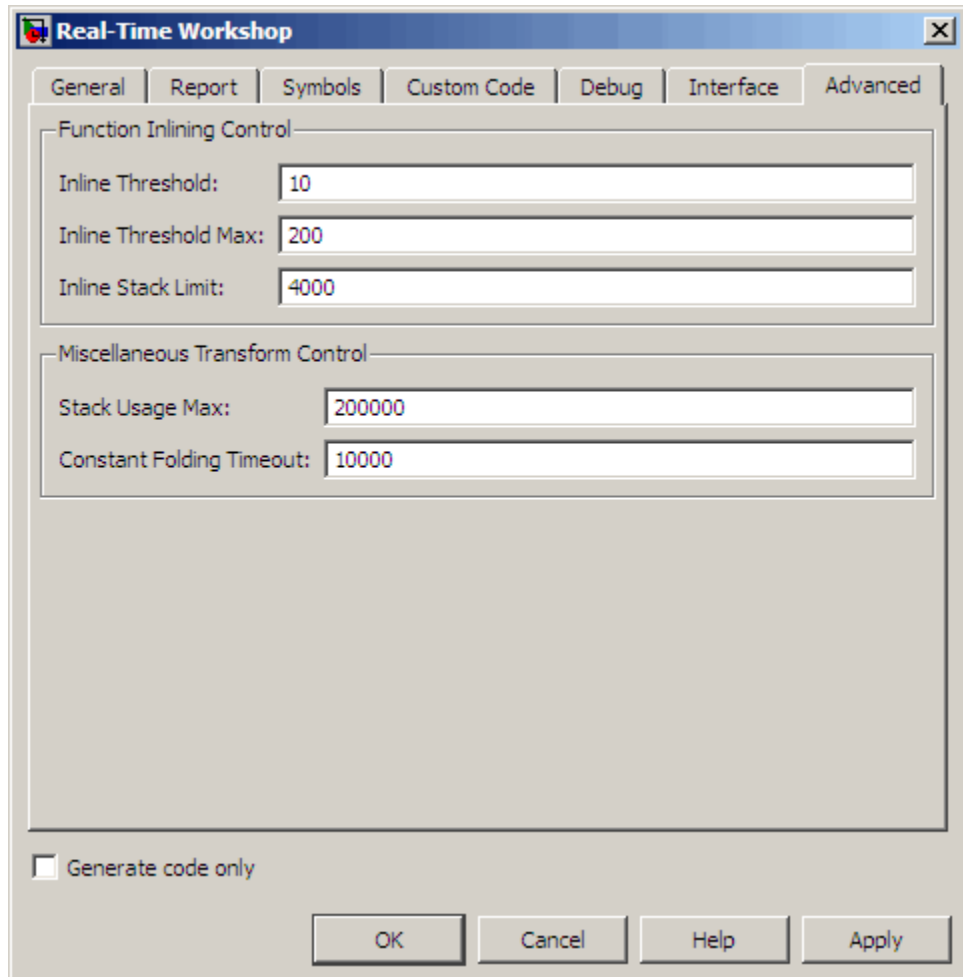
The following table describes the interface parameters for the Embedded MATLAB Coder **Real-Time Workshop** dialog box:

| Interface Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---|---|--|
| “Target function library” on page 6-146 | TargetFunctionLibrary <i>string</i> , ANSI_C | <p>Specify a target-specific math library for your model.</p> <p>Supports target function libraries (TFLs) for GRT system target files.</p> <p>If you have a Real-Time Workshop Embedded Coder license, you can configure Embedded MATLAB Coder to use ERT TFLs when generating C code. You enable this feature by defining a configuration object for C code generation using an <code>ert</code> parameter at the MATLAB command prompt, as in this example:</p> <pre>rtwcfg = emlcoder.RTWConfig('ert')</pre> |
| “Support: floating-point numbers” on page 6-150 | PurelyIntegerCode <i>string</i> , on | <p>Specify whether to generate floating-point data and operations.</p> <p>Dependencies:</p> <ul style="list-style-type: none"> • This parameter only appears for ERT-based targets. • This parameter requires a Real-Time Workshop Embedded Coder license when generating code. • Selecting this parameter enables Support: non-finite numbers and clearing this parameter disables Support: non-finite numbers. |
| “Support: non-finite numbers” on page 6-154 | SupportNonFinite <i>string</i> , on | <p>Specify whether to generate nonfinite data and operations.</p> <p>Dependencies:</p> |

| Interface Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---------------------|---|---|
| | | <ul style="list-style-type: none"> • This parameter only appears for ERT-based targets. • This parameter requires a Real-Time Workshop Embedded Coder license when generating code. • This parameter is enabled by Support: floating-point numbers. |

Advanced Tab

Specifies parameters for fine-tuning the behavior of the compiler.



Parameters

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

| Advanced Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---------------------------|--|---|
| Inline Threshold | InlineThreshold <i>integer, 10</i> | Specify the maximum size of functions to be inlined. |
| Inline Threshold Max | InlineThresholdMax <i>integer, 200</i> | Specify the maximum size of functions after inlining. |
| Inline Stack Limit | InlineStackLimit <i>integer, 4000</i> | Specify the stack size limit on inlined functions. |
| Stack Usage Max | StackUsageMax <i>integer, 200000</i> | Specify the maximum stack usage per function. |
| Constant Folding Timeout | ConstantFoldingTimeout <i>integer, 10000</i> | Specify the maximum number of instructions to be executed by the constant folder. |

Generate code only

Specify code generation versus an executable build. See “Generate code only” on page 6-34.

Automatic C MEX Generation Dialog Box for Embedded MATLAB Coder

In this section...

“Automatic C MEX Generation Dialog Box Overview” on page 7-25

“General Tab” on page 7-26

“Report Tab” on page 7-30

“Symbols Tab” on page 7-32

“Custom Code Tab” on page 7-34

“Advanced Tab” on page 7-37

Automatic C MEX Generation Dialog Box Overview

Specifies parameters for C MEX generation using Embedded MATLAB Coder.

Displaying the Dialog Box

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

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

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

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

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

```
open mexgen_cfg;
```

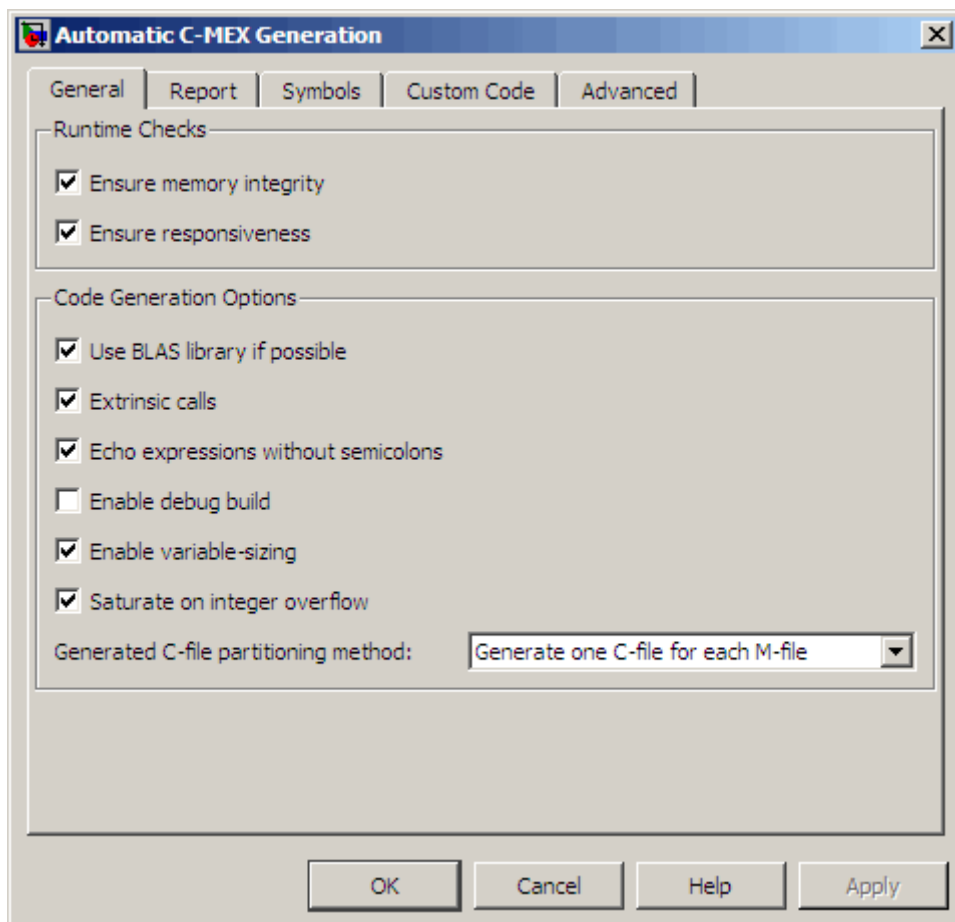
The dialog box displays on your desktop.

See Also

“Configuring Your Environment for Code Generation”

General Tab

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



Parameters

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

| General Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|--|---|--|
| Ensure memory integrity | IntegrityChecks true , false | Detect violations of memory integrity in code generated for Embedded MATLAB functions and stops execution with a diagnostic message. Setting IntegrityChecks to false also disables the runtime stack. |
| Ensure responsiveness | ResponsivenessChecks true , false | Enable responsiveness checks in code generated for Embedded MATLAB functions. |
| Use BLAS library if possible | EnableBLAS true , false | Speed up low-level matrix operations during simulation by calling the Basic Linear Algebra Subprograms (BLAS) library. |
| Extrinsic calls | ExtrinsicCalls true , false | Allow calls out to MATLAB. When disabled, attempts to remove extrinsic calls. |
| “Echo expressions without semicolons” on page 7-28 | EchoExpressions true , false | Specify whether or not actions that do not terminate with a semicolon appear in the MATLAB Command Window. |
| Enable debug build | EnableDebugging true, false | Compile the generated code in debug mode. |
| Enable variable-sizing | EnableVariableSizing true , false | Enable support for variable-sized arrays. |

| General Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|--------------------------------------|--|--|
| Saturate on integer overflow | SaturateOnIntegerOverflow true , false | Add checks in the generated code to detect integer overflow or underflow. |
| Generated C-file partitioning method | FilePartitionMethod <i>string</i> , MapMFileToCFile , SingleFile | Specify whether to generate one C-file for each M-file or generate all C functions into a single file. |

See Also.

- “Speeding Up Simulation with the Basic Linear Algebra Subprograms (BLAS) Library” in the *Real-Time Workshop User’s Guide*.
- “Controlling Runtime Checks” in the *Real-Time Workshop User’s Guide*.
- “Generating Code for Variable-Size Data” in the *Embedded MATLAB User’s Guide*.

Echo expressions without semicolons

For C MEX code generation, specify whether or not actions that do not terminate with a semicolon appear in the MATLAB Command Window.

Settings. Default: true

true
Enables output to appear in the MATLAB Command Window for actions that do not terminate with a semicolon.

false
Disables output from appearing in the MATLAB Command Window for actions that do not terminate with a semicolon.

Command-Line Information.

Parameter: EchoExpressions
Type: boolean

Value: true | false

Default: true

Recommended Settings.

| Application | Setting |
|-------------------|-----------|
| Debugging | true |
| Traceability | No impact |
| Efficiency | false |
| Safety precaution | No impact |

Enable debug build

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

Settings. Default: false



true

Compile generated code in debug mode.



false

Compile generated code in release (or optimized) mode.

Command-Line Information.

Parameter: EnableDebugging

Type: boolean

Value: true | false

Default: false

Recommended Settings.

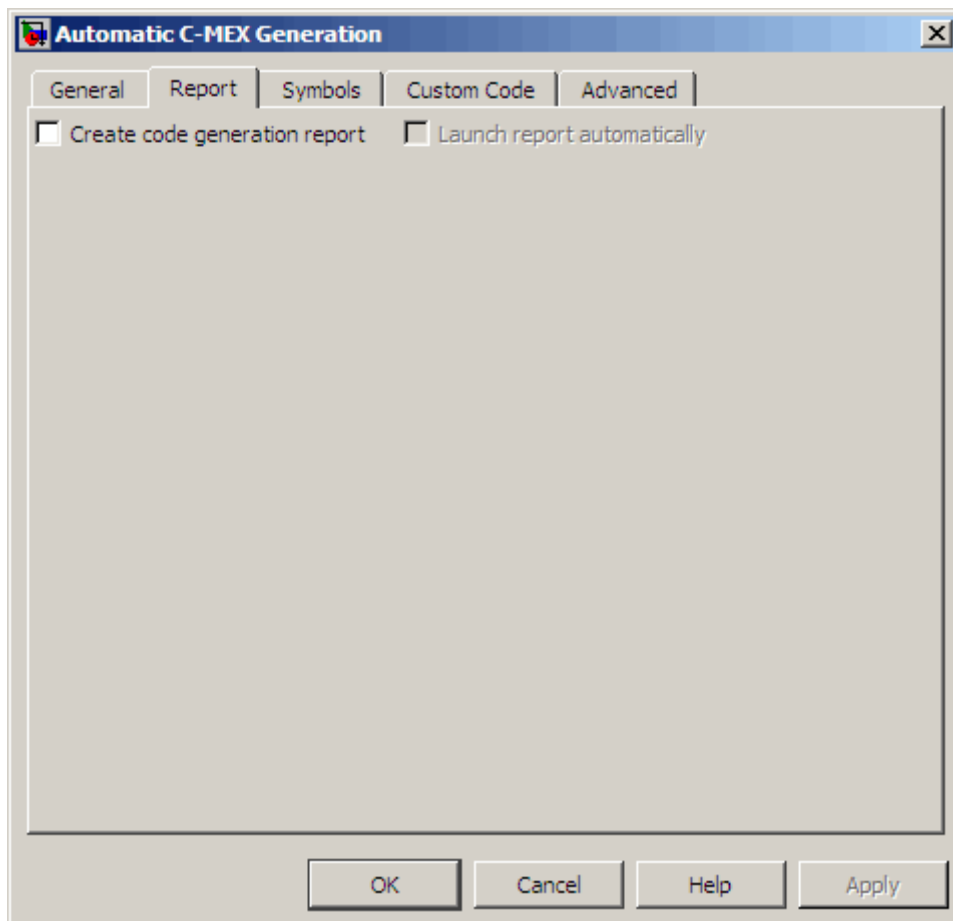
| Application | Setting |
|--------------|---------|
| Debugging | true |
| Traceability | true |

| Application | Setting |
|-------------------|-----------|
| Efficiency | false |
| Safety precaution | No impact |

See Also. “How Debugging Affects Simulation Speed” in the Simulink User’s Guide.

Report Tab

Controls the report that is created for C MEX generation using Embedded MATLAB Coder.



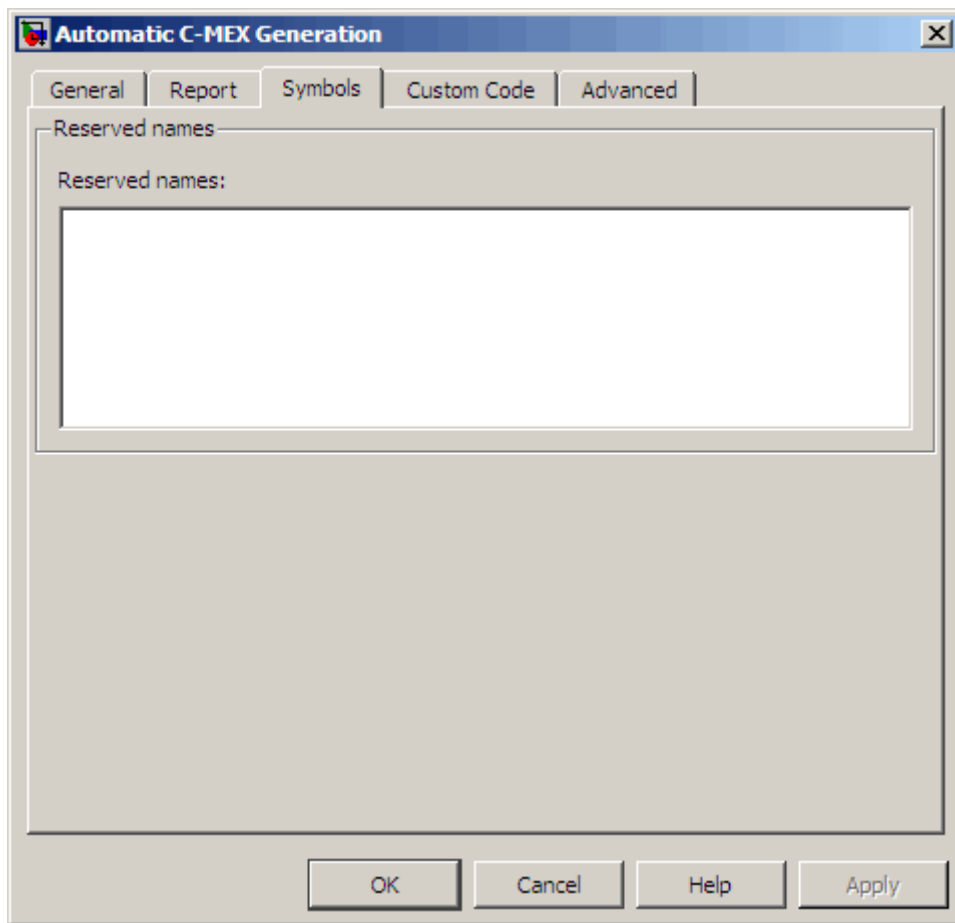
Parameters

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

| Report Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|--|--|--|
| “Create code generation report” on page 6-40 | GenerateReport true, false | Document generated code in an HTML report. |
| “Launch report automatically” on page 6-43 | LaunchReport true, false | <p>Specify whether to automatically display HTML reports after code generation completes.</p> <hr/> <p>Note Requires that you select Create code generation report</p> <hr/> |

Symbols Tab

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



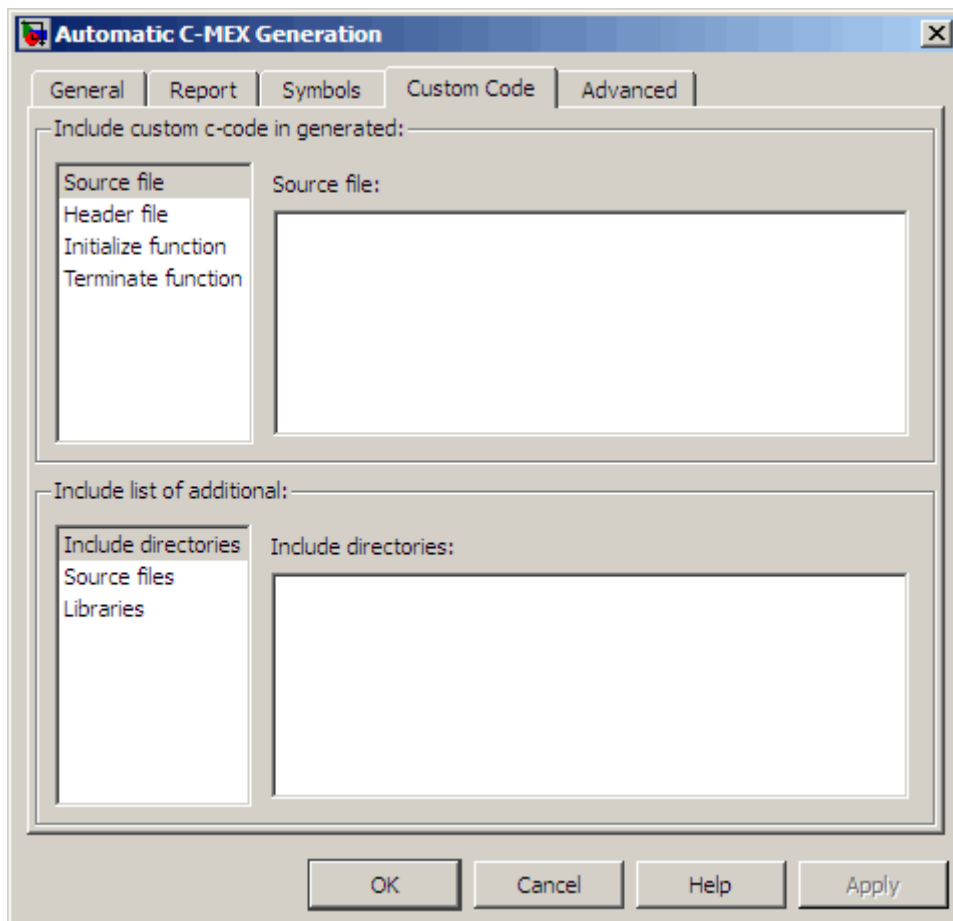
Parameters

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

| Symbols Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|--------------------------------|--|--|
| “Reserved names” on page 6-114 | ReservedNameArray <i>string</i> , | Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code. |

Custom Code Tab

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



Configuration

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

Parameters

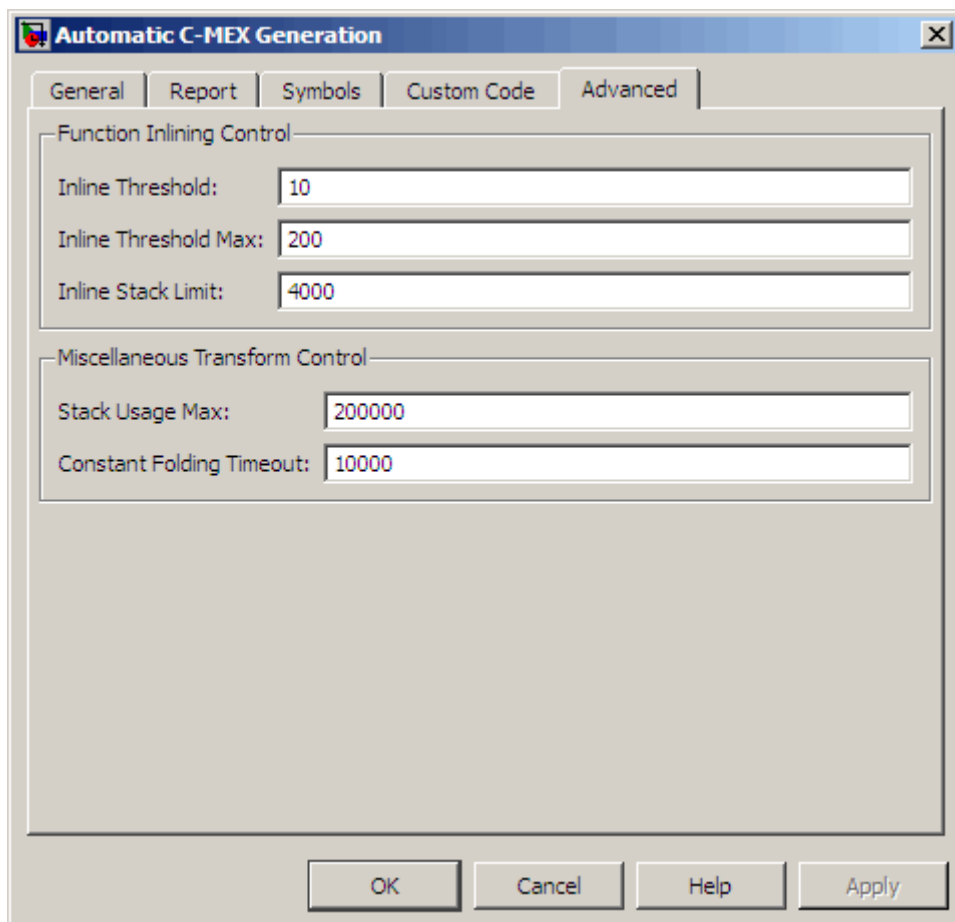
The following table describes the custom code parameters for the Embedded MATLAB Coder Automatic C MEX Generation dialog box:

| Custom Code Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|-------------------------------------|---|--|
| “Source file” on page 6-122 | CustomSourceCode <i>string</i> , | Specify code appearing near the top of the generated C MEX file. |
| “Header file” on page 6-123 | CustomHeaderCode <i>string</i> , | Specify code appearing near the top of the generated header .h file. |
| “Initialize function” on page 6-124 | CustomInitializer <i>string</i> , | Specify code appearing in the initialize function of the generated C MEX file. |
| “Terminate function” on page 6-125 | CustomTerminator <i>string</i> , | Specify code appearing in the terminate function of the generated .c or .cpp file. |
| “Include directories” on page 6-126 | CustomInclude <i>string</i> , | <p>Specify a space-separated list of include directories to add to the include path when compiling the generated code.</p> <hr/> <p>Note If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example,</p> <pre>'C:\Project "C:\Custom Files"'</pre> <hr/> |

| Custom Code Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|------------------------------|---|--|
| “Source files” on page 6-128 | CustomSource <i>string</i> , | <p>Specify a space-separated list of source files to compile and link with the generated code.</p> <hr/> <p>Note If your list includes any Windows file name strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example,</p> <pre>'algs.c "div alg.c"'</pre> <hr/> |
| “Libraries” on page 6-130 | CustomLibrary <i>string</i> , | <p>Specify a space-separated list of static library files to link with the generated code.</p> <hr/> <p>Note If your list includes any Windows file name strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example,</p> <pre>'mathutil.lib "op utils.lib"'</pre> <hr/> |

Advanced Tab

Specifies parameters for fine-tuning the behavior of the compiler.



Parameters

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

| Advanced Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---------------------------|--|--|
| Inline Threshold | InlineThreshold <i>integer, 10</i> | Specify the maximum size of functions to be inlined. |

| Advanced Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---------------------------|--|---|
| Inline Threshold Max | InlineThresholdMax <i>integer, 200</i> | Specify the maximum size of functions after inlining. |
| Inline Stack Limit | InlineStackLimit <i>integer, 4000</i> | Specify the stack size limit on inlined functions. |
| Stack Usage Max | StackUsageMax <i>integer, 200000</i> | Specify the maximum stack usage per function. |
| Constant Folding Timeout | ConstantFoldingTimeout <i>integer, 10000</i> | Specify the maximum number of instructions to be executed by the constant folder. |

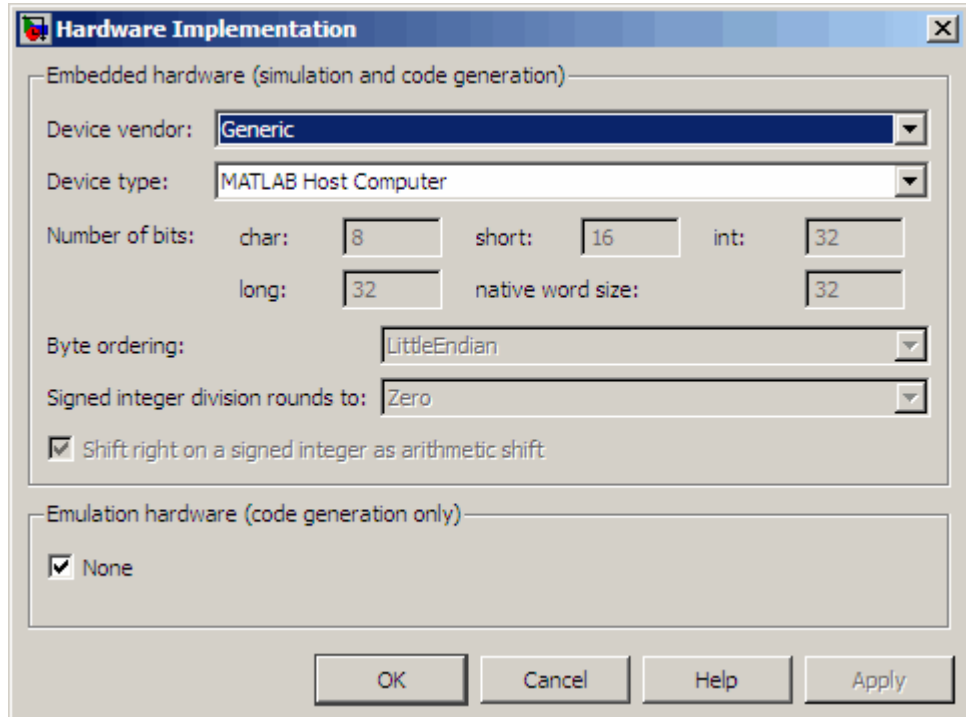
Hardware Implementation Dialog Box for Embedded MATLAB Coder

| In this section... |
|---|
| “Hardware Implementation Parameters Dialog Box Overview” on page 7-40 |
| “Hardware Implementation Parameters” on page 7-42 |

Hardware Implementation Parameters Dialog Box Overview

Specifies parameters of the target hardware implementation.

The Hardware Implementation parameters dialog box appears as follows by default. (The **long** and **native word size** values shown are for a 32-bit MATLAB host computer.) In this case, the test hardware and the deployment hardware are the same and you use the **Embedded hardware** pane to set the hardware properties for both.



Displaying the Dialog Box

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

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

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

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

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

```
open hwi_cfg;
```

The dialog box displays on your desktop.

See Also

“Configuring Your Environment for Code Generation”

Hardware Implementation Parameters

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

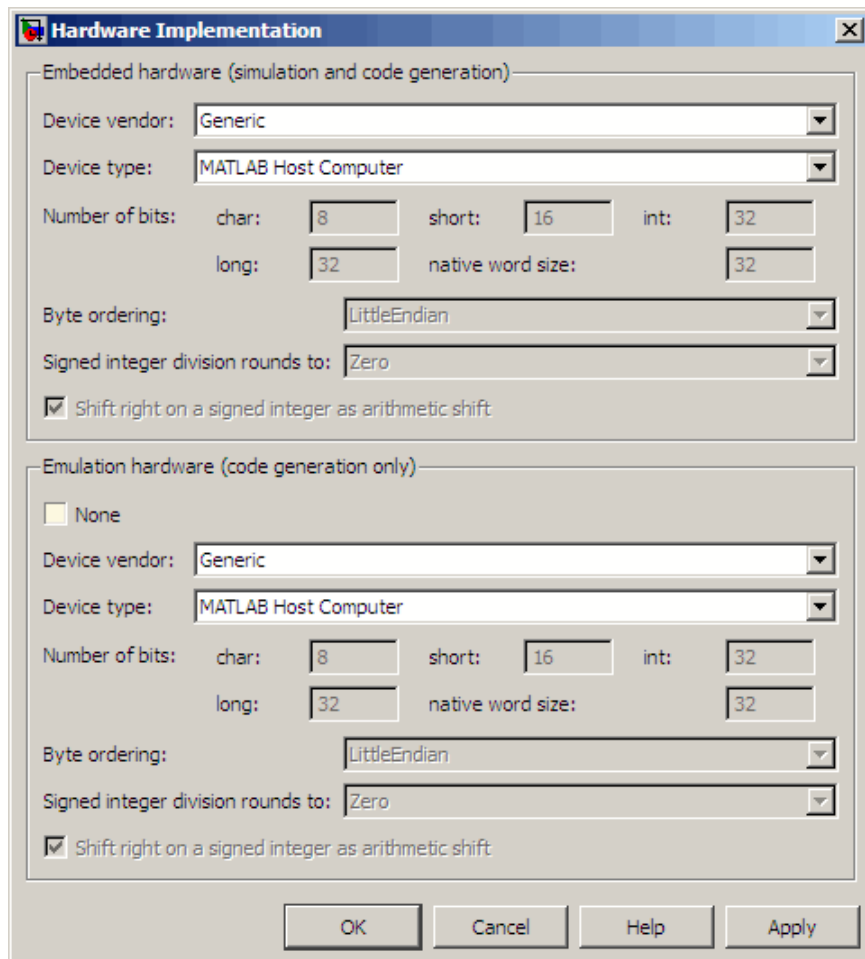
| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|-------------------------|---|--|
| “Device vendor” | ProdHWDeviceType <i>string</i> , Generic->MATLAB Host Computer | Specify manufacturer of hardware you will use to implement the production version of the system. |
| “Device type” | ProdHWDeviceType <i>string</i> , Generic->MATLAB Host Computer | Specify type of hardware you will use to implement the production version of the system. |
| “Number of bits: char” | ProdBitPerChar <i>integer</i> , 8 | Describe length in bits of the C char data type supported by the deployment hardware. |
| “Number of bits: short” | ProdBitPerShort <i>integer</i> , 16 | Describe length in bits of the C short data type supported by the deployment hardware. |
| “Number of bits: int” | ProdBitPerInt <i>integer</i> , 32 | Describe length in bits of the C int data type supported by the deployment hardware. |

| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---|--|--|
| “Number of bits: long” | ProdBitPerLong <i>integer</i> , host-specific value (32 or 64) | Describe length in bits of the C long data type supported by the deployment hardware. |
| “Number of bits: native word size” | ProdWordSize <i>integer</i> , host-specific value (32 or 64) | Describe microprocessor native word size for the deployment hardware. |
| “Byte ordering” | ProdEndianess 'Unspecified', LittleEndian , 'BigEndian' | Describe significance of the first byte of a data word for the deployment hardware. |
| “Signed integer division rounds to” | ProdIntDivRoundTo 'Undefined', Zero , 'Floor' | Describe how your compiler rounds the result of dividing one signed integer by another to produce a signed integer quotient. |
| “Shift right on a signed integer as arithmetic shift” | ProdShiftRightIntArith true , false | Describe whether your compiler implements a signed integer right shift as an arithmetic right shift. |
| None | ProdEqTarget true , false | Specify whether the test hardware differs from the deployment hardware. <hr/> Note By default they are the same. <hr/> Dependencies: • Setting this parameter to true disables the |

| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|------------------|--|--|
| | | <p>Emulation hardware (code generation only) subpane.</p> <ul style="list-style-type: none"> Setting this parameter to false enables the Emulation hardware (code generation only) subpane. This subpane is used to specify the test hardware properties. <p>See “Emulation Hardware Subpane” on page 7-45</p> |

Emulation Hardware Subpane

The Hardware Implementation parameters dialog box displays the **Emulation hardware (code generation only)** subpane when the test hardware and the deployment hardware differ. In this case, you use the **Embedded hardware (simulation and code generation)** subpane to specify the deployment hardware properties and the **Emulation hardware (code generation only)** pane to specify the test hardware properties.



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

| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|-------------------------|---|---|
| “Device vendor” | TargetHWDeviceType <i>string</i> , Generic->MATLAB Host Computer | Select the manufacturer of the hardware that will be used to test the generated code. Dependency: • None is set to false |
| “Device type” | TargetHWDeviceType <i>string</i> , Generic->MATLAB Host Computer | Select the type of hardware that will be used to test the generated code. Dependency: • None is set to false |
| “Number of bits: char” | TargetBitPerChar <i>integer</i> , 8 | Describe length in bits of the C char data type supported by the test hardware. Dependency: • None is set to false |
| “Number of bits: short” | TargetBitPerShort <i>integer</i> , 16 | Describe length in bits of the C short data type supported by the test hardware. Dependency: • None is set to false |

| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|------------------------------------|--|---|
| “Number of bits: int” | TargetBitPerInt <i>integer</i> , 32 | Describe length in bits of the C int data type supported by the test hardware. Dependency: • None is set to false |
| “Number of bits: long” | TargetBitPerLong <i>integer</i> , host-specific value (32 or 64) | Describe length in bits of the C long data type supported by the test hardware. Dependency: • None is set to false |
| “Number of bits: native word size” | TargetWordSize <i>integer</i> , host-specific value (32 or 64) | Describe the microprocessor native word size for the test hardware. Dependency: • None is set to false |
| “Byte ordering” | TargetEndianness 'Unspecified', LittleEndian , 'BigEndian' | Describe significance of the first byte of a data word for the test hardware. Dependency: • None is set to false |

| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|---|--|--|
| “Signed integer division rounds to” | TargetIntDivRoundTo 'Undefined', Zero , 'Floor' | Describe how to produce a signed integer quotient for the test hardware. Dependency: • None is set to false |
| “Shift right on a signed integer as arithmetic shift” | TargetShiftRightIntArith true , false | Describe how your compiler rounds the result of two signed integers for the test hardware. Dependency: • None is set to false |

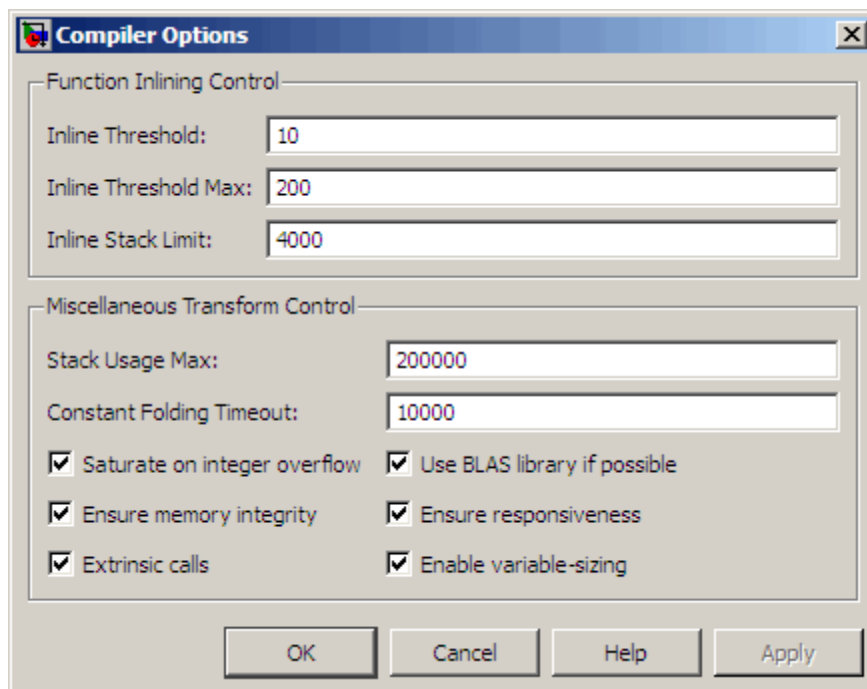
Compiler Options Dialog Box

| In this section... |
|--|
| “Compiler Options Parameters Dialog Box Overview” on page 7-49 |
| “Compiler Options Parameters” on page 7-51 |

Compiler Options Parameters Dialog Box Overview

Note This dialog is for use with the Embedded MATLAB function `emlmex` only. It is available for use with Embedded MATLAB Coder for backwards compatibility purposes only and will be removed in a future release. To specify parameters for embeddable C code generation using Embedded MATLAB Coder, use the “Real-Time Workshop Dialog Box for Embedded MATLAB Coder” on page 7-2. To specify parameters for C MEX generation using Embedded MATLAB Coder, use the “Automatic C MEX Generation Dialog Box for Embedded MATLAB Coder” on page 7-25.

Specifies parameters for fine-tuning the behavior of the compiler.



Displaying the Dialog Box

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

- 1 Define a configuration object variable for compiler options in the MATLAB workspace by issuing a constructor command like this:

```
co_cfg=emlcoder.CompilerOptions;
```

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

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

```
open co_cfg;
```


The dialog box displays on your desktop.

Compiler Options Parameters

The following table describes the parameters for fine-tuning the behavior of the compiler for Embedded MATLAB Coder:

| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|------------------------------|--|--|
| Inline Threshold | <code>InlineThreshold</code> <i>integer</i> , 10 | Specify the maximum size of functions to be inlined. |
| Inline Threshold Max | <code>InlineThresholdMax</code> <i>integer</i> , 200 | Specify the maximum size of functions after inlining. |
| Inline Stack Limit | <code>InlineStackLimit</code> <i>integer</i> , 4000 | Specify the stack size limit on inlined functions. |
| Stack Usage Max | <code>StackUsageMax</code> <i>integer</i> , 200000 | Specify the maximum stack usage per function. |
| Constant Folding Timeout | <code>ConstantFoldingTimeout</code> <i>integer</i> , 10000 | Specify the maximum number of instructions to be executed by the constant folder. |
| Saturate on integer overflow | <code>SaturateOnIntegerOverflow</code> true , false | Add checks in the generated code to detect integer overflow or underflow. |
| Use BLAS library if possible | <code>EnableBLAS</code> true , false | Speed up low-level matrix operations during simulation by calling the Basic Linear Algebra Subprograms (BLAS) library. |
| Ensure memory integrity | <code>IntegrityChecks</code> true , false | Detect violations of memory integrity in code generated for Embedded MATLAB functions and stops execution with a diagnostic message. |

| Parameter | Equivalent Command-Line Property and Values (default in bold) | Description |
|------------------------|--|---|
| Ensure responsiveness | ResponsivenessChecks true , false | Enable responsiveness checks in code generated for Embedded MATLAB functions. |
| Extrinsic calls | ExtrinsicCalls true , false | Allow calls out to MATLAB. When disabled, attempts to remove extrinsic calls. |
| Enable variable-sizing | EnableVariableSizing true , false | Enable support for variable-sized arrays. |

Model Advisor Checks

Real-Time Workshop Checks

| In this section... |
|---|
| “Real-Time Workshop Overview” on page 8-3 |
| “Check solver for code generation” on page 8-4 |
| “Identify questionable blocks within the specified system” on page 8-6 |
| “Check for model reference configuration mismatch” on page 8-7 |
| “Check the hardware implementation” on page 8-8 |
| “Identify questionable software environment specifications” on page 8-10 |
| “Identify questionable code instrumentation (data I/O)” on page 8-12 |
| “Check for blocks that have constraints on tunable parameters” on page 8-13 |
| “Identify questionable subsystem settings” on page 8-15 |
| “Disable signal logging” on page 8-16 |
| “Identify blocks that generate expensive saturation and rounding code” on page 8-17 |
| “Check sample times and tasking mode” on page 8-18 |
| “Identify questionable fixed-point operations” on page 8-19 |
| “Check model configuration settings against code generation objectives” on page 8-27 |

Real-Time Workshop Overview

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

See Also

- [Consulting Model Advisor](#)
- [Simulink Model Advisor Check Reference](#)
- [Simulink Verification and Validation Model Advisor Check Reference](#)

Check solver for code generation

Check model solver and sample time configuration settings.

Description

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

Results and Recommended Actions

| Condition | Recommended Action |
|---|--|
| The solver type is set incorrectly for model level code generation. | Set Configuration Parameters > Solver > <ul style="list-style-type: none"> • Type to Fixed-step • Solver to Discrete (no continuous states) |
| Multitasking diagnostic options are not set to error. | Set Configuration Parameters > Diagnostics > <ul style="list-style-type: none"> • Sample Time > Multitask conditionally executed subsystem to error • Sample Time > Multitask rate transition to error • Data Validity > Multitask data store to error |

Tips

You do not have to modify the solver settings to generate code from a subsystem. The Real-Time Workshop Embedded Coder build process automatically changes **Solver type** to fixed-step when you select **Real-Time Workshop > Build Subsystem** or **Real-Time Workshop > Generate S-Function** from the subsystem context menu.

See Also

- “Configuring Scheduling”
- “Executing Multitasking Models”

Identify questionable blocks within the specified system

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

Description

The code generator creates code only for the blocks that it supports. Some blocks are not recommended for production code deployment.

Results and Recommended Actions

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

See Also

“Block Support Considerations”

Check for model reference configuration mismatch

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

Description

The code generator cannot create code for top models that contain referenced models with different, incompatible configuration parameter settings.

Results and Recommended Actions

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

See Also

Model Referencing Configuration Parameter Requirements

Check the hardware implementation

Identify inconsistent or underspecified hardware implementation settings

Description

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

Results and Recommended Actions

| Condition | Recommended Action |
|--|---|
| Your system target file is <code>grt.tlc</code> . | Use an ERT-based target to generate final production code. |
| Hardware implementation parameters are not set to recommended values. | Specify the following Configuration Parameters > Hardware Implementation parameters to the recommended values: <ul style="list-style-type: none"> • Number of bits • Byte ordering • Signed integer division rounding |
| Hardware implementation Embedded Hardware settings do not match Emulation Hardware settings. | Select the Configuration Parameters > Hardware Implementation > None check box and configure the Emulation hardware settings. |
| The target hardware has not been configured. | Specify the parameters in the Emulation hardware box in the Configuration Parameters > Hardware Implementation pane. |

Limitations

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

See Also

Making GRT-Based Targets ERT-Compatible

Identify questionable software environment specifications

Identify questionable software environment settings.

Description

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

Results and Recommended Actions

| Condition | Recommended Action |
|---|---|
| The maximum identifier length does not conform with industry standards for C. | Set the Configuration Parameters > Real-Time Workshop > Symbols > Maximum identifier length parameter to 31 characters. |
| Real-Time Workshop Interface parameters are not set to recommended values. | Set the following Configuration Parameters > Real-Time Workshop > Interface parameters to the recommended values: <ul style="list-style-type: none"> • Support: continuous time • Support: non-finite numbers • Support: non-inlined S-functions |
| Real-Time Workshop Symbols parameters are not set to recommended values. | Set the Configuration Parameters > Real-Time Workshop > Symbols > Generate scalar inlined parameters as parameter to Literals. |

| Condition | Recommended Action |
|--|---|
| <p>Support: variable-size signals is selected. This might lead to inefficient code.</p> | <p>If you do not intend to support variable-sized signals, clear the Real-Time Workshop > Interface > “Support: variable-size signals” on page 6-162 check box in the Configuration Parameters dialog box.</p> |
| <p>The model contains Stateflow charts with weak Simulink I/O data type specifications.</p> | <p>Select the Stateflow chart property Use Strong Data Typing with Simulink I/O. You might need to adjust the data types in your model after selecting the property.</p> |

Limitations

A Stateflow license is required when using Stateflow charts.

See Also

“Strong Data Typing with Simulink I/O”

Identify questionable code instrumentation (data I/O)

Identify questionable code instrumentation.

Description

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

Results and Recommended Actions

| Condition | Recommended Action |
|---|--|
| Interface parameters are not set to recommended values. | Set the Configuration Parameters > Real-Time Workshop > Interface parameters to the recommended values. |
| Blocks generate assertion code. | Set the Configuration Parameters > Diagnostics > Data Validity > Model Verification block enabling parameter to Disable All on a block-by-block basis or globally. |
| Block output signals have one or more test points and the Ignore test point signals check box is cleared in the Real-Time Workshop pane of the Configuration Parameters dialog box. | Remove test points from the specified block output signals. For each signal, in the Signal Properties dialog box, clear the Test point check box. Alternatively, if the model is using an ERT-based system target file, select the Ignore test point signals check box in the Real-Time Workshop pane of the Configuration Parameters dialog box to ignore test points during code generation. |

Check for blocks that have constraints on tunable parameters

Identify blocks with constraints on tunable parameters.

Description

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

Results and Recommended Actions

| Condition | Recommended Action |
|--|--|
| Lookup Table blocks have tunable parameters. | <p>When tuning parameters during simulation or when running the generated code, you must:</p> <ul style="list-style-type: none"> • Preserve monotonicity of the setting for the Vector of input values parameter. • Preserve the number and location of zero values that you specify for Vector of input values and Vector of output values parameters if you specify multiple zero values for the Vector of input values parameter. |
| Lookup Table (2-D) blocks have tunable parameters. | <p>When tuning parameters during simulation or when running the generated code, you must:</p> <ul style="list-style-type: none"> • Preserve monotonicity of the setting for the Row index input values and Column index of input values parameters. • Preserve the number and location of zero values that you specify for Row index input values, Column index of input values, |

| Condition | Recommended Action |
|-----------|--|
| | and Vector of output values parameters if you specify multiple zero values for the Row index input values or Column index of input values parameters. |

See Also

Lookup Table block

Identify questionable subsystem settings

Identify questionable subsystem block settings.

Description

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

Results and Recommended Actions

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

See Also

Subsystem block

Disable signal logging

Disables unnecessary signal logging.

Description

Disabling unnecessary signal logging avoids declaring extra signal memory in generated code.

Analysis Results and Recommended Actions

| Conditions | Recommended Action |
|---------------------|--|
| Signals are logged. | Disable signal logging on all signals. |

Action Results

Clicking **Modify All** disables signal logging on all logged signals.

Identify blocks that generate expensive saturation and rounding code

Check for blocks that generate expensive saturation or rounding code.

Description

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

Results and Recommended Actions

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

Check sample times and tasking mode

Set up the sample time and tasking mode for your system.

Description

Incorrect tasking mode can result in inefficient code execution or incorrect generated code.

Results and Recommended Actions

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

See Also

“Single-Tasking and Multitasking Execution Modes”

Identify questionable fixed-point operations

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

Description

The following operations can lead to nonoptimal results:

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

- Blocks with relative scaling of inputs and outputs must determine the ideal product in the generated C code. The C code required to do this multiplication is large and slow.
- Blocks that multiply signals with nonzero bias require extra steps to implement the multiplication. Inserting Data Type Conversion blocks remove the biases, and allow you to control data type and scaling for intermediate calculations. The conversion is done once and all blocks in the subsystem benefit from simpler, bias-free math.
- Blocks are multiplying signals with mismatched slope adjustment factors. This mismatch causes the overall operation to involve two multiply instructions.
- Blocks are multiplying signals with mismatched slope adjustment factors. This mismatch causes the overall operation to involve integer multiplication followed by shifts. Under certain simplicity and accuracy conditions when the net slope is a reciprocal of an integer, it is sometimes more efficient to replace the multiplication and shifts with an integer division.
- the Real-Time Workshop software generates a reciprocal operation followed by a multiply operation for Product blocks that have a divide operation for the first input, and a multiply operation for the second input. If you reverse the inputs so that the multiplication occurs first and the division occurs second, the Real-Time Workshop software generates a single division operation for both inputs.
- An input with an invariant constant value is used as the denominator in an online division operation. If the operation is changed to multiplication, and the invariant input is replaced by its reciprocal, then the division is done offline and the online operation is multiplication. This leads to faster and smaller generated code.
- Addition
 - Sum blocks can have a range error when the input range exceeds the output range.
 - A Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output. This mismatch requires the Sum block to do one or more multiplication operations.

- The net sum of the Sum block input biases does not equal the bias of the output. The generated code includes one extra addition or subtraction instruction to correctly account for the net bias adjustment. For better accuracy and efficiency, nonzero bias terms are collected into a single net bias correction term. The ranges given for the input and output exclude their biases.
- Using Relational Operator blocks
 - The data types of the Relational Operator block inputs are not the same. A conversion operation is required every time the block is executed. If one of the inputs is invariant, then changing the data type and scaling of the invariant input to match the other input improves the efficiency of the model.
 - The Relational Operator block inputs have different ranges, resulting in a range error when casting, and a precision loss each time a conversion is performed. You can insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type that has sufficient range and precision to represent each input, making the relational operation error-free.
 - The inputs of the Relational Operator block have different slope adjustment factors. The mismatch causes the Relational Operator block to require a multiply operation each time the input with lesser positive range is converted to the data type and scaling of the input with greater positive range.
- Using MinMax blocks
 - The input and output of the MinMax block have different data types. A conversion operation is required every time the block is executed. The model is more efficient with the same data types.
 - The input of the MinMax block is converted to the data type and scaling of the output before performing a relational operation, resulting in a range error when casting, or a precision loss each time a conversion is performed.
 - The input of the MinMax block has a different slope adjustment factor than the output. This mismatch causes the MinMax block to require a multiply operation each time the input is converted to the data type and scaling of the output.

- Discrete-Time Integrator blocks have a complicated initial condition setting. The initial condition for the Discrete-Time Integrator blocks are used to initialize the state and output. As a result, the output equation generates excessive code and an extra global variable is required.

Results and Recommended Actions

| Condition | Recommended Action |
|---|---|
| Integer division generated code is large. | Set the Configuration Parameters > Hardware Implementation > Signed integer division rounds to parameter to the recommended value. |
| Protection code generated as part of the division operation is redundant. | Verify that your model cannot cause exceptions in division operations and then remove redundant protection code by setting the Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions parameter. |
| Generated code is inefficient. | Set the Function Block Parameters > Integer rounding mode parameter to the recommended value. |
| Lookup Table vector of input values is not evenly spaced. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_look1_func_approx</code> . |
| Lookup Table vector of input values is not evenly spaced when quantized, but it is very close to being evenly spaced. | If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_evenspace_cleanup</code> . |

| Condition | Recommended Action |
|--|--|
| <p>Lookup Table vector of input values is evenly spaced, but the spacing is not a power of 2.</p> | <p>If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_look1_func_approx</code>.</p> |
| <p>For a Prelookup or Lookup Table (n-D) block, Index search method is Evenly spaced points. Breakpoint data does not have power of 2 spacing.</p> | <p>If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. Otherwise, in the block parameter dialog box, specify a different Index search method to avoid the computation-intensive division operation.</p> |
| <p>Lookup Table (n-D) breakpoint data is not evenly spaced and Index search method is not Evenly spaced points.</p> | <p>If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing and then set Index search method to Evenly spaced points.</p> |
| <p>Lookup Table (n-D) breakpoint data is evenly spaced and Index search method is Evenly spaced points. But the spacing is not a power of 2.</p> | <p>If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_look1_func_approx</code>.</p> |
| <p>Lookup Table (n-D) breakpoint data is evenly spaced, but the spacing is not a power of 2. Also, Index search method is not Evenly spaced points.</p> | <p>Set Index search method to Evenly spaced points. Also, if the data is nontunable, consider an even, power of 2 spacing.</p> |
| <p>Lookup Table (n-D) breakpoint data is evenly spaced, and the spacing is a power of 2. But the Index search method is not Evenly spaced points.</p> | <p>Set Index search method to Evenly spaced points.</p> |
| <p>Blocks require cumbersome multiplication.</p> | <p>Restrict multiplication operations:</p> <ul style="list-style-type: none"> • So the product integer size is no larger than the target integer size. • To the recommended size. |

| Condition | Recommended Action |
|---|---|
| <p>Blocks multiply signals with nonzero bias.</p> | <p>Insert a Data Type Conversion block before and after the block containing the multiplication operation.</p> |
| <p>Product blocks are multiplying signals with mismatched slope adjustment factors.</p> | <p>Change the scaling of the output so that its slope adjustment factor is the product of the input slope adjustment factor.</p> |
| <p>Product blocks are multiplying signals with mismatched slope adjustment factors. The net slope correction uses multiplication followed by shifts, which is inefficient for some target hardware.</p> | <p>Select Use integer division to handle net slopes that are reciprocals of integers if the net slope is the reciprocal of an integer and division is more efficient than multiplication and shifts on the target hardware.</p> |
| | <hr/> <p>Note This optimization takes place only if certain simplicity and accuracy conditions are met. For more information, see Use Integer Division for Net Slope Correction in the Simulink Fixed Point documentation.</p> <hr/> |
| <p>Product blocks are configured to do multiple division operations.</p> | <p>Multiply all the denominator terms together, and then do a single division using cascading Product blocks.</p> |
| <p>Product blocks are configured to do many multiplication or division operations.</p> | <p>Split the operations across several blocks, with each block performing one multiplication or one division operation.</p> |

| Condition | Recommended Action |
|---|---|
| <p>Product blocks are configured with a divide operation for the first input and a multiply operation for the second input.</p> | <p>Reverse the inputs so the multiply operation occurs first and the division operation occurs second.</p> |
| <p>An input with an invariant constant value is used as the denominator in an online division operation.</p> | <p>Change the operation to multiplication, and replace the invariant input by its reciprocal.</p> |
| <p>The data type range of the inputs of Sum blocks exceeds the data type range of the output, which can cause overflow or saturation.</p> | <p>Change the output and accumulator data types so the range equals or exceeds all input ranges.</p> <p>For example, if the model has two inputs:</p> |
| | <ul style="list-style-type: none"> • int8 (–128 to 127) • uint8 (0 to 255) <p>The data type range of the output and accumulator must equal or exceed –128 to 255. A int16 (–32768 to 32767) data type meets this condition.</p> |
| <p>A Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output.</p> | <p>Change the data types so the inputs, outputs, and accumulator have the same slope adjustment factor.</p> |
| <p>The net sum of the Sum block input biases does not equal the bias of the output.</p> | <p>Change the bias of the output scaling, making the net bias adjustment zero.</p> |

| Condition | Recommended Action |
|--|---|
| The inputs of the Relational Operator block have different data types. | <ul style="list-style-type: none"> • Change the data type and scaling of the invariant input to match other inputs. • Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type. |
| The inputs of the Relational Operator block have different slope adjustment factor. | Change the scaling of either input. |
| The input and output of the MinMax block have different data types. | Change the data type of the input or output. |
| The input of the MinMax block has a different slope adjustment factor than the output. | Change the scaling of the input or the output. |
| The initial condition of the Discrete-Time Integrator block is used to initialize both the state and the output. | Set the Function Block Parameters > Use initial condition as initial and reset value for parameter to State only (most efficient). |

Limitations

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

See Also

- Lookup Table
- Lookup Table (n-D)
- Prelookup
- Remove code that protects against division arithmetic exceptions

Check model configuration settings against code generation objectives

Check the configuration parameter settings for the model against the code generation objectives.

Description

Each parameter in the Configuration Parameters dialog box might have different recommended settings for code generation based on your objectives. This check helps you identify the recommended setting for each parameter so that you can achieve optimized code based on your objective.

Analysis Results and Recommended Actions

| Condition | Recommended Action |
|---|--|
| Parameters are set to values other than the value recommended for the specified objectives. | Set the parameters to the recommended values. <hr/> Note A change to one parameter value can impact other parameters. Successfully passing the check might take multiple iterations. |

Action Results

Clicking **Modify Parameters** changes the parameter values to the recommended values.

See Also

- The Real-Time Workshop “Recommended Settings Summary” on page 6-261
- The Real-Time Workshop Embedded Coder “Recommended Settings Summary”
- “Configuring Code Generation Objectives” in the Real-Time Workshop User’s Guide.

- “Mapping Application Objectives to Model Configuration Parameters” in the Real-Time Workshop Embedded Coder documentation.

A

- addCompileFlags function 3-2
- addDefines function 3-6
- addIncludeFiles function 3-9
- addIncludePaths function 3-13
- addLinkFlags function 3-16
- addLinkObjects function 3-19
- addNonBuildFiles function 3-24
- addSourceFiles function 3-27
- addSourcePaths function 3-31
- addTMFTokens function 3-34
- Async Interrupt block 5-2
- automatic C MEX generation parameters
 - Embedded MATLAB Coder 7-25

B

- blocks
 - Async Interrupt 5-2
 - Model Header
 - reference 5-8
 - Model Source
 - reference 5-9
 - Protected RT 5-10
 - RTW S-Function 5-11
 - System Derivatives 5-13
 - System Disable 5-14
 - System Enable 5-16
 - reference 5-15
 - System Outputs 5-17
 - System Start 5-18
 - System Terminate 5-19
 - System Update 5-20
 - Task Sync 5-21
 - Unprotected RT 5-25

C

- C MEX generation parameters
 - Embedded MATLAB Coder

- advanced 7-37
 - custom code 7-34
 - general 7-26
 - report 7-30
 - symbols 7-32
- General pane
 - Echo expressions without semicolons 7-28
 - Enable debug build 7-29
- code generation parameters
 - Embedded MATLAB Coder 7-2
 - advanced 7-23
 - custom code 7-13
 - debug 7-16
 - general 7-3
 - generate code only 7-24
 - interface 7-18
 - report 7-6
 - symbols 7-7
- compiler options
 - adding to build information 3-2
- compiler options parameters
 - Embedded MATLAB Coder 7-49
- configuration parameters
 - code generation 6-285
 - impacts of settings 6-261
- pane
 - Generate preprocessor conditionals 6-194
 - Global types: 6-92
 - variable-size signals 6-162
- Real-Time Workshop (general)
 - Check model before generating code 6-32
- Configuration Parameters
 - Real-Time Workshop (general)
 - Check model 6-31
- Configuration Parameters dialog box
 - Real-Time Workshop (comments)
 - Comments tab overview 6-60
 - Custom comments 6-69

- Custom comments function 6-71
- Include comments 6-61
- Requirements in block comments 6-75
- Show eliminated blocks 6-63
- Simulink block descriptions 6-65
- Simulink block Stateflow object comments 6-62
- Simulink data object descriptions 6-67
- Stateflow object descriptions 6-73
- Verbose comments for Simulink global storage class 6-64
- Real-Time Workshop (custom code)
 - Custom Code tab overview 6-118
 - Header file 6-123
 - Include directories 6-126
 - Initialize function 6-124
 - Libraries 6-130
 - Source file 6-122
 - Source files 6-128
 - Terminate function 6-125
 - Use local custom code settings (do not inherit from main model) 6-120
 - Use the same custom code settings as Simulation Target 6-119
- Real-Time Workshop (debug)
 - Debug tab overview 6-134
 - Enable TLC assertion 6-140
 - Profile TLC 6-137
 - Retain .rtw file 6-136
 - Start TLC coverage when generating code 6-139
 - Start TLC debugger when generating code 6-138
 - Verbose build 6-135
- Real-Time Workshop (general)
 - Build/Generate code 6-36
 - Compiler optimization level 6-10
 - Custom compiler optimization flags 6-12
 - General tab overview 6-5
 - Generate code only 6-34
 - Generate makefile 6-15
 - Ignore custom storage classes 6-21
 - Ignore test point signals 6-23
 - Language 6-8
 - Make command 6-17
 - Prioritized objectives code 6-27
 - Select objective 6-25
 - Set objectives 6-28
 - System target file 6-6
 - Template makefile 6-19
 - TLC options 6-13
- Real-Time Workshop (interface)
 - Block parameter access 6-184
 - Block parameter visibility 6-180
 - Configure C++ Encapsulation Interface 6-199
 - Configure Model Functions 6-198
 - Create Simulink (S-Function) block 6-200
 - Enable portable word sizes 6-202
 - External I/O access 6-188
 - Generate destructor 6-190
 - Generate reusable code 6-173
 - GRT compatible call interface 6-167 interface 6-208
 - Interface tab overview 6-145
 - Internal data access 6-186
 - Internal data visibility 6-182
 - MAT-file logging 6-204
 - MAT-file variable name modifier 6-206
 - Maximum word length 6-165
 - MEX-file arguments 6-215
 - Multiword type definitions 6-163
 - Parameters in C API 6-212
 - Pass root-level I/O as 6-178
 - Reusable code error diagnostic 6-176
 - Signals in C API 6-211
 - Single output/update function 6-169
 - Static memory allocation 6-217
 - Static memory buffer size 6-219

- Support absolute time 6-152
- Support complex numbers 6-158
- Support continuous time 6-156
- Support floating-point numbers 6-150
- Support non-finite numbers 6-154
- Support non-inlined S-functions 6-160
- Suppress error status in real-time model
 - data structure 6-196
- Target function library 6-146
- Terminate function required 6-171
- Transport layer 6-213
- Use operator new for referenced model
 - object registration 6-192
- Utility function 6-148
- Real-Time Workshop (Real-Time Workshop S-Function Code Generation Options)
 - Create new model 6-230
 - Include custom source code 6-232
 - Real-Time Workshop S-Function
 - Code Generation Options Tab
 - Overview 6-229
 - Use value for tunable parameters 6-231
- Real-Time Workshop (report)
 - Code-to-model 6-45
 - Configure 6-49
 - Create code generation report 6-40
 - Eliminated / virtual blocks 6-50
 - Launch report automatically 6-43
 - Model-to-code 6-47
 - Report tab overview 6-39
 - Traceable Embedded MATLAB
 - functions 6-56
 - Traceable Simulink blocks 6-52
 - Traceable Stateflow objects 6-54
- Real-Time Workshop (RSim Target)
 - Enable RSim executable to load
 - parameters from a MAT-file 6-224
 - Force storage classes to AUTO 6-226
 - RSim Target tab overview 6-223
- Solver selection 6-225
- Real-Time Workshop (symbols)
 - Constant macros 6-98
 - #define naming 6-111
 - Field name of global types 6-87
 - Generate scalar inlined parameter
 - as 6-104
 - Global types 6-84
 - Global variables 6-82
 - Local block output variables 6-96
 - Local temporary variables 6-94
 - M-function 6-107
 - Maximum identifier length 6-102
 - Minimum mangle length 6-100
 - Parameter naming 6-109
 - Reserved names 6-114
 - Signal naming 6-105
 - Subsystem methods 6-89
 - Symbols tab overview 6-81
 - Use the same reserved names as
 - Simulation Target 6-113
- Real-Time Workshop (Tornado Target)
 - Base task priority 6-248
 - Code format 6-243
 - Download to VxWorks target 6-246
 - External mode 6-251
 - MAT-file logging 6-239
 - MAT-file variable name modifier 6-241
 - MEX-file arguments 6-255
 - Static memory allocation 6-257
 - Static memory buffer size 6-259
 - StethoScope 6-244
 - Target function library 6-236
 - Task stack size 6-250
 - Tornado Target tab overview 6-235
 - Transport layer 6-253
 - Utility function 6-238
- Configuration Set Objectives dialog box 6-29

D

- debugging
 - and configuration parameter settings 6-261
- derivatives
 - in custom code 5-13
- disable code
 - in custom code 5-14

E

- efficiency
 - and configuration parameter settings 6-261
- Embedded MATLAB Coder
 - automatic C MEX generation
 - parameters 7-25
 - C MEX generation parameters
 - advanced 7-37
 - custom code 7-34
 - Echo expressions without semicolons 7-28
 - Enable debug build 7-29
 - general 7-26
 - report 7-30
 - symbols 7-32
 - code generation parameters 7-2
 - advanced 7-23
 - custom code 7-13
 - debug 7-16
 - general 7-3
 - generate code only 7-24
 - interface 7-18
 - report 7-6
 - symbols 7-7
 - compiler options parameters 7-49
 - hardware implementation parameters 7-40
- enable code
 - in custom code 5-15
- extensions, file. *See* file extensions

F

- file extensions
 - updating in build information 3-96
- file separator
 - changing in build information 3-99
- file types. *See* file extensions
- findIncludeFiles function 3-47

G

- getCompileFlags function 3-49
- getDefines function 3-51
- getFullFileList function 3-55
- getIncludeFiles function 3-56
- getIncludePaths function 3-59
- getLinkFlags function 3-61
- getNonBuildFiles function 3-64
- getSourceFiles function 3-66
- getSourcePaths function 3-69

H

- hardware implementation parameters
 - Embedded MATLAB Coder 7-40
- header files
 - finding for inclusion in build information
 - object 3-47

I

- include files
 - adding to build information 3-9
 - finding for inclusion in build information
 - object 3-47
 - getting from build information 3-56
- include paths
 - adding to build information 3-13
 - getting from build information 3-59
- initialization code
 - in custom code 5-16

interrupt service routines
 creating 5-2

L

limitations
 of Real-Time Workshop product 1-1
link objects
 adding to build information 3-19
link options
 adding to build information 3-16
 getting from build information 3-61

M

macros
 defining in build information 3-6
 getting from build information 3-51
makefile
 generating and executing for system 3-49
model header
 in custom code 5-8
Model Header block
 reference 5-8
Model Source block
 reference 5-9
models
 parameters for configuring 6-285

N

nonbuild files
 adding to build information 3-24
 getting from build information 3-64

O

outputs code
 in custom code 5-17

P

packNGo function 3-72
parameter structure
 getting 3-74
parameters
 for configuring model code generation and
 targets 6-285
paths
 updating in build information 3-96
project files
 packaging for relocation 3-72
Protected RT block 5-10

R

rate transitions
 protected 5-10
 unprotected 5-25
RSim target
 parameter loading 6-224
rsimgetrtpt function 3-74
RTW S-Function block 5-11
RTW.getBuildDir function 3-84

S

S-function target
 generating 5-11
safety precautions
 and configuration parameter settings 6-261
separator, file
 changing in build information 3-99
source code
 in custom code 5-9
source files
 adding to build information 3-27
 getting from build information 3-66
source paths
 adding to build information 3-31
 getting from build information 3-69

- startup code
 - in custom code 5-18
- System Derivatives block 5-13
- System Disable block 5-14
- System Enable block 5-15
- System Initialize block 5-16
- System Outputs block 5-17
- System Start block 5-18
- System Terminate block 5-19
- System Update block 5-20

T

- Target Language Compiler and Function Library 3-93
- targets
 - parameters for configuring 6-285
- task function

- creating 5-21
- Task Sync block 5-21
- termination code
 - in custom code 5-19
- t1c function 3-93
- TMF tokens
 - adding to build information 3-34
- traceability
 - and configuration parameter settings 6-261

U

- Unprotected RT block 5-25
- update code
 - in custom code 5-20
- updateFilePathsAndExtensions function 3-96
- updateFileSeparator function 3-99