Simulink® Coder™

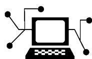
Reference

R2011b

MATLAB® & SIMULINK®

MathWorks®

How to Contact 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.

Simulink® Coder™ Reference

© COPYRIGHT 2011 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

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

Revision History

April 2011	Online only	New for Version 8.0 (Release 2011a)
September 2011	Online only	Revised for Version 8.1 (Release 2011b)

Contents

1 Product Limitations Summary

Glossary

2 Function Reference

Build Information **2-2**

Build Process **2-4**

Desktop IDEs and Desktop Targets **2-6**
IDE Automation Interface **2-6**
XMakefile **2-7**

Project Documentation **2-8**

Rapid Simulation **2-9**

Target Language Compiler and Function Library **2-10**

3 Alphabetical List

4 Block Reference

Asynchronous **4-2**
Interrupt Templates **4-2**

Custom Code **4-3**

Desktop Targets (desktoptargetslib) **4-4**
Host Communication **4-4**
Target Preferences **4-4**
Linux **4-4**
Windows **4-5**

S-Function Target **4-6**

5 Blocks — Alphabetical List

6 Configuration Parameters for Simulink Models

Code Generation Pane: General **6-2**
Code Generation: 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
Set Objectives — Code Generation Advisor Dialog Box . . . 6-29
Check model . . . 6-32
Check model before generating code . . . 6-33
Generate code only . . . 6-35
Build/Generate code . . . 6-37

Code Generation Pane: Report . . . 6-38
Code Generation: Report Tab Overview . . . 6-40
Create code generation report . . . 6-41
Launch report automatically . . . 6-44
Code-to-model . . . 6-46
Model-to-code . . . 6-48
Configure . . . 6-50
Eliminated / virtual blocks . . . 6-51
Traceable Simulink blocks . . . 6-53
Traceable Stateflow objects . . . 6-55
Traceable MATLAB functions . . . 6-57
Static code metrics . . . 6-59

Code Generation Pane: Comments . . . 6-61
Code Generation: Comments Tab Overview . . . 6-63
Include comments . . . 6-64
Simulink block / Stateflow object comments . . . 6-66
MATLAB source code as comments . . . 6-67
Show eliminated blocks . . . 6-69
Verbose comments for SimulinkGlobal storage class . . . 6-70
Simulink block descriptions . . . 6-71
Simulink data object descriptions . . . 6-73
Custom comments (MPT objects only) . . . 6-75
Custom comments function . . . 6-77
Stateflow object descriptions . . . 6-79
Requirements in block comments . . . 6-81
MATLAB function help text . . . 6-83

Code Generation Pane: Symbols . . . 6-85

Code Generation: Symbols Tab Overview **6-88**
Global variables **6-89**
Global types **6-91**
Field name of global types **6-94**
Subsystem methods **6-96**
Subsystem method arguments **6-99**
Local temporary variables **6-101**
Local block output variables **6-103**
Constant macros **6-105**
Minimum mangle length **6-107**
Maximum identifier length **6-109**
Generate scalar inlined parameter as **6-111**
Signal naming **6-112**
M-function **6-114**
Parameter naming **6-116**
#define naming **6-118**
Use the same reserved names as Simulation Target **6-120**
Reserved names **6-121**

Code Generation Pane: Custom Code **6-123**
Code Generation: Custom Code Tab Overview **6-126**
Use the same custom code settings as Simulation Target .. **6-127**
Use local custom code settings (do not inherit from main model) **6-128**
Source file **6-130**
Header file **6-131**
Initialize function **6-132**
Terminate function **6-133**
Include directories **6-134**
Source files **6-136**
Libraries **6-138**

Code Generation Pane: Debug **6-140**
Code Generation: Debug Tab Overview **6-142**
Verbose build **6-143**
Retain .rtw file **6-144**
Profile TLC **6-145**
Start TLC debugger when generating code **6-146**
Start TLC coverage when generating code **6-148**
Enable TLC assertion **6-149**

Code Generation Pane: Interface **6-150**
Code Generation: Interface Tab Overview **6-154**

Target function library **6-155**
Custom **6-158**
Shared code placement **6-159**
Support: floating-point numbers **6-161**
Support: non-finite numbers **6-163**
Support: complex numbers **6-165**
Support: absolute time **6-166**
Support: continuous time **6-168**
Support: non-inlined S-functions **6-170**
Support: variable-size signals **6-172**
Multiword type definitions **6-173**
Maximum word length **6-175**
GRT compatible call interface **6-177**
Single output/update function **6-179**
Terminate function required **6-181**
Generate reusable code **6-183**
Reusable code error diagnostic **6-186**
Pass root-level I/O as **6-188**
Block parameter visibility **6-190**
Internal data visibility **6-192**
Block parameter access **6-194**
Internal data access **6-196**
External I/O access **6-198**
Generate destructor **6-200**
Use operator new for referenced model object registration **6-202**
Generate preprocessor conditionals **6-204**
Suppress error status in real-time model data structure **6-206**
Combine signal/state structures **6-208**
Configure Model Functions **6-211**
Configure C++ Encapsulation Interface **6-212**
MAT-file logging **6-213**
MAT-file variable name modifier **6-216**
Interface **6-218**
Generate C API for: signals **6-221**
Generate C API for: parameters **6-222**
Generate C API for: states **6-223**
Generate C API for: root-level I/O **6-224**
Transport layer **6-225**
MEX-file arguments **6-227**
Static memory allocation **6-229**
Static memory buffer size **6-231**

Code Generation Pane: RSim Target **6-233**

Code Generation: RSim Target Tab Overview **6-235**
Enable RSim executable to load parameters from a MAT-file **6-236**
Solver selection **6-237**
Force storage classes to AUTO **6-238**

Code Generation Pane: S-Function Target **6-239**
Code Generation S-Function Target Tab Overview **6-241**
Create new model **6-242**
Use value for tunable parameters **6-243**
Include custom source code **6-244**

Code Generation Pane: Tornado Target **6-245**
Code Generation: Tornado Target Tab Overview **6-247**
Target function library **6-248**
Shared code placement **6-250**
MAT-file logging **6-252**
MAT-file variable name modifier **6-254**
Code Format **6-256**
StethoScope **6-257**
Download to VxWorks target **6-259**
Base task priority **6-261**
Task stack size **6-263**
External mode **6-264**
Transport layer **6-266**
MEX-file arguments **6-268**
Static memory allocation **6-270**
Static memory buffer size **6-272**

Code Generation Pane: IDE Link **6-274**
Overview **6-276**
Build format **6-277**
Build action **6-279**
Overrun notification **6-282**
Function name **6-284**
Configuration **6-285**
Compiler options string **6-287**
Linker options string **6-289**
System stack size (MAUs) **6-291**
Profile real-time execution **6-294**
Profile by **6-296**
Number of profiling samples to collect **6-298**
Maximum time allowed to build project (s) **6-300**

Maximum time allowed to complete IDE operation (s) **6-302**
Export IDE link handle to base workspace **6-303**
IDE link handle name **6-305**
Source file replacement **6-306**

Parameter Reference **6-308**
Recommended Settings Summary **6-308**
Parameter Command-Line Information Summary **6-337**

7 Model Advisor Checks

Embedded Coder Checks **7-2**
Checks Overview **7-3**
Check solver for code generation **7-4**
Identify questionable blocks within the specified system .. **7-6**
Identify lookup table blocks that generate expensive out-of-range checking code **7-7**
Check output types of logic blocks **7-9**
Identify blocks using one-based indexing **7-10**
Check the hardware implementation **7-11**
Identify questionable software environment specifications **7-12**
Identify questionable code instrumentation (data I/O) **7-14**
Check for blocks that have constraints on tunable parameters **7-15**
Check for blocks not recommended for MISRA-C:2004 compliance **7-17**
Check configuration parameters for MISRA-C:2004 compliance **7-18**
Check for model reference configuration mismatch **7-20**
Identify blocks that generate expensive saturation and rounding code **7-21**
Check sample times and tasking mode **7-22**
Identify questionable subsystem settings **7-23**
Identify questionable fixed-point operations **7-24**
Check model configuration settings against code generation objectives **7-33**
Check for efficiency optimization parameters **7-34**

Index

Product Limitations Summary

The following topics identify Simulink® Coder™ feature limitations:

- "C++ Target Language Limitations"
- "packNGo Function Limitations"
- "Tunable Expression Limitations"
- "Limitations on Specifying Data Types in the Workspace Explicitly"
- "Code Reuse Limitations"
- "Simulink Coder Model Referencing Limitations"
- "External Mode Limitations"
- "Noninlined S-Function Parameter Type Limitations"
- "S-Function Target Limitations"
- "Rapid Simulation Target Limitations"
- "Asynchronous Support Limitations"
- "C API Limitations"
- "Supported Products and Block Usage"

Glossary

application modules
: With respect to Simulink Coder 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 Simulink Coder 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 Simulink Coder 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 Simulink Coder 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 **Code Generation** 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 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 Simulink Coder 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`	Simulink Coder	Makefile specific to your model that is derived from the template makefile
`.rtw`	Simulink Coder	Intermediate compilation (*`model`*`.rtw`) of a `.mdl` file used in generating C or C++ code
`.tlc`	MathWorks and Simulink Coder users	Target Language Compiler script files that the Simulink Coder build process uses to generate code for targets and blocks
`.tmf`	Supplied with Simulink Coder	Template makefiles
`.tmw`	Simulink Coder	Project marker file inside a build folder 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 Simulink Coder 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 `.tlc` 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 Simulink Coder 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 Simulink Coder 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 Simulink Coder 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 Simulink Coder 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.

Simulink Coder target

Set of code files generated by the Simulink Coder build process for a standard or custom target, specified by a Simulink Coder 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 MATLAB® code. The Simulink Coder 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 Simulink Coder 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 folder in the working folder. Also an executable library compiled from these codes that implements a Model block. *See also* Simulink Coder 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 Simulink Coder 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 Simulink Coder 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 Simulink Coder 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
Desktop IDEs and Desktop Targets (p. 2-6)	Control IDEs and software build tool chains for desktop targets
Project Documentation (p. 2-8)	Document generated code
Rapid Simulation (p. 2-9)	Get model's parameter structures
Target Language Compiler and Function Library (p. 2-10)	Optimize code generated for model's blocks

Build Information

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

getIncludePaths	Include paths from model's build information
getLinkFlags	Link options from model's build information
getNonBuildFiles	Nonbuild-related files from model's build information
getSourceFiles	Source files from model's build information
getSourcePaths	Source paths from model's build information
packNGo	Package model code in zip file for relocation
updateFilePathsAndExtensions	Update files in model's build information with missing paths and file extensions
updateFileSeparator	Change file separator used in model's build information

Build Process

RTW.getBuildDir	Build folder information for specified model
rtwbuild	Initiate build process
rtwrebuild	Rebuild generated code
rtw_precompile_libs	Build libraries within model without building model
Simulink.fileGenControl	Specify root folders in which to put files generated by diagram updates and model builds
switchTarget	Specify target for configuration set

Desktop IDEs and Desktop Targets

In this section...
"IDE Automation Interface" on page 2-6
"XMakefile" on page 2-7

IDE Automation Interface

Eclipse IDE

activate	Mark file, project, or build configuration as active
add	Add files to active project in IDE
address	Memory address and page value of symbol in IDE
build	Build or rebuild current project
close	Close project in IDE window
dir	Files and folders in current IDE window
display (IDE Object)	Properties of IDE handle
eclipseide	Create handle object to interact with Eclipse IDE
eclipseidesetup	Configure your coder product to interact with Eclipse IDE
halt	Halt program execution by processor
insert	Insert debug point in file
isrunning	Determine whether processor is executing process
load	Load program file onto processor
new	Create project, library, or build configuration in IDE

open	Open project in IDE
pwd	Working folder used by Eclipse™
read	Read data from processor memory
reload	Reload most recent program file to processor signal processor
remove	Remove file, project, or breakpoint
restart	Reload most recent program file to processor signal processor
run	Execute program loaded on processor
write	Write data to processor memory block
xmakefilesetup	Configure your coder product to generate makefiles

XMakefile

xmakefilesetup	Configure your coder product to generate makefiles

Project Documentation

rtwreport	Generate report documenting generated code for 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	Invoke Target Language Compiler to convert model description file to generated code

See the “TLC Function Library Reference” in the Simulink Coder 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 one or more compiler options
- Multiple group names to collections of compiler options (available for nonmakefile build environments only)

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

Note

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

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

addCompileFlags

Description

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

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

Examples

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

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

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

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

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

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

See Also

addDefines | addLinkFlags | getCompileFlags

How To

- “Customizing Post-Code-Generation Build Processing”

Purpose

Mark file, project, or build configuration as active

Syntax

IDE_Obj.activate('*objectname*','*type*')

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

Use the *IDE_Obj*.activate('*objectname*','*type*') method to make a project file or build configuration active in the MATLAB session.

When you make a project, file, or build configuration active, methods you invoke on the IDE handle object apply to that project, file, or build configuration.

Input Arguments

IDE_Obj

For *IDE_Obj*, enter the name of the IDE handle object you created using a constructor function.

objectname

For *objectname*, enter the name of the project file or build configuration to make active.

For project files, enter the full file name including the extension.

For build configurations, enter 'Debug', 'Release', or 'Custom'. Before using the activate method on a build configuration, activate the project that contains the build configuration. For more information about configurations, see "Configuration" on page 6-285.

type

For *type*, enter the type of object to make active. If you omit the *type* argument, *type* defaults to 'project'. Enter one of the following strings for *type*:

- 'project' — Makes a specified project active.

- `'buildcfg'` — Make a specified build configuration active

IDE support for *type*

	CCS	Eclipse	MULTI	VisualDSP++
'project'	Yes	Yes	Yes	Yes
'buildcfg'	Yes	Yes		Yes

Examples

After using a constructor to create the IDE handle object, `h`, open several projects, make the first one active, and build the project:

```
h.open('c:\temp\myproj1')
h.open('c:\temp\myproj2')
h.open('c:\temp\myproj3')
h.activate('c:\temp\myproj1', 'project')
h.build
```

After making a project active, make the `'debug'` configuration active:

```
h.activate('debug','buildcfg')
```

See Also

`build` | `new` | `remove`

Purpose

Add files to active project in IDE

Syntax

IDE_Obj.add(*filename*,*filetype*)

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

Use *IDE_Obj*.add(*filename*,*filetype*) to add an existing file to the active project in the IDE. Using the add function is equivalent to selecting **Project > Add Files to Project** in the IDE.

Before using add:

- Use the constructor function for your IDE to create an IDE handle object, such as *IDE_Obj*.
- Create or open a project using the new or open methods.
- Make the project active in the IDE using the activate method.

You can add any file type your IDE supports to your project. Consult the documentation for your IDE for detailed information about supported file types.

All Supported File Types and Extensions

File Type	Extensions Supported	CCS IDE Project Folder
C/C++ source files	.c, .cpp, .cc, .cxx, .sa, .h,.hpp,.hxx	Source
Assembly source files	.a*, .s* (excluding .sa), .dsp	Source
Object and library files	.o*, .lib, .doj, .dlb	Libraries
Linker command file	.cmd, .ldf	Project Name

All Supported File Types and Extensions (Continued)

File Type	Extensions Supported	CCS IDE Project Folder
VDK support file	`.vdk`	Not applicable
DSP/BIOS file (only with CCS IDE)	`.tcf`	`DSP/BIOS Config`

Note CCS IDE drops files in the appropriate project folder, indicated in the right-most column of the preceding table.

Input Arguments

`add` places the file specified by *`filename`* in the active project in the IDE.

`IDE_Obj`

`IDE_Obj` is a handle for an instance of the IDE. Before using a method, the constructor function for your IDE to create *`IDE_Obj`*.

`filename`

`filename` is the name of the file to add to the active IDE project.

If you supply a filename with no path or with a relative path, your coder product searches the IDE working folder first. It then searches the folders on your MATLAB path. Add supported file types shown in the preceding table.

`filetype`

`filetype` is an optional argument that specifies the file type. For example, `'lib'`, `'src'`, `'header'`.

Examples

Start by creating an IDE handle object, such as `IDE_Obj` using the constructor for your IDE. Then enter the following commands:

```
IDE_Obj.new('myproject','project');  % Create a new project.
```

```
IDE_Obj.add('sourcefile.c'); % Add a C source file.
```

See Also

activate | | new | open | remove

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 one or more macro definitions
- Multiple group names to collections of macro definitions (available for nonmakefile build environments only)

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

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

Description

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

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

Examples

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

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

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

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

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

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

See Also

`addCompileFlags` | `addLinkFlags` | `getDefines`

How To

- "Customizing Post-Code-Generation Build Processing"

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

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

You can apply

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

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

Description

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

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

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

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

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

Examples

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

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

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

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

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

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

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

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

See Also

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

How To

- “Customizing Post-Code-Generation Build Processing”

Purpose

Add include paths to model's build information

Syntax

`addIncludePaths(buildinfo, paths, groups)`

groups is optional.

Arguments

buildinfo

Build information returned by `RTW.BuildInfo`.

paths

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

The function removes duplicate include file entries that

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

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

groups (optional)

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

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

You can apply

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

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

Description

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

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

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

Examples

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

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

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

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

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

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

See Also

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

How To

- “Customizing Post-Code-Generation Build Processing”

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 one or more linker options
- Multiple group names to collections of linker options (available for nonmakefile build environments only)

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

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

Description

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

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

Examples

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

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

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

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

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

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

See Also

`addCompileFlags` | `addDefines` | `getLinkFlags`

How To

- "Customizing Post-Code-Generation Build Processing"

Purpose

Add link objects to model's build information

Syntax

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

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

Arguments

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

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

  The function removes duplicate link objects that

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

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

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

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

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

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

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

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

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

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

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

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 of elements you specify for *linkobjs*.

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

Description

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

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

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

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

If You Specify *priority*, *precompiled*, or *linkonly* as a...	**The Function...**
Value	Applies the value to all objects it adds to the build information.
Vector of values	Pairs each value with a specified object. Thus, the length of the vector must match the length of the cell array you specify for *linkobjs*.

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

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

Examples

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

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

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

- Add the linkable objects `libobj1` and `libobj2` to build information `myModelBuildInfo`. Mark both objects as 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');
```

How To

- “Customizing Post-Code-Generation Build Processing”

Purpose

Add nonbuild-related files to model's build information

Syntax

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

`paths` and *`groups`* are optional.

Arguments

`buildinfo`

Build information returned by `RTW.BuildInfo`.

`filenames`

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

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

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

The function removes duplicate nonbuild file entries that

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

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

`paths` (optional)

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

`groups` (optional)

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

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

You can apply

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

To...	Specify *groups* as a...
Apply one group name to 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 *filenames*.

Description

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

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

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to 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 *filenames*.

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

Examples

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

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

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

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

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

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

See Also

getNonBuildFiles

How To

- “Customizing Post-Code-Generation Build Processing”

Purpose

Memory address and page value of symbol in IDE

Syntax

`a = IDE_Obj.address(symbol,scope)`

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

The `a = IDE_Obj.address(symbol,scope)` method returns the memory address of the first matching symbol in the symbol table of the most recently loaded program.

Because the `address` method returns the `address` and `page` values as a structure, your programs can use the values directly. For example, the `IDE_Obj.read` and `IDE_Obj.write` can use `a` as an input.

If the address method does not find the symbol in the symbol table, it generates a warning and returns a null value.

Input Arguments

`a`

Use *a* as a variable to capture the return values from the `address` method.

`IDE_Obj`

IDE_Obj is a handle for an instance of the IDE. Before using a method, use the constructor function for your IDE to create *IDE_Obj*.

`symbol`

symbol is the name of the symbol for which you are getting the memory address and page values.

Symbol names are case sensitive. Use the proper case when you enter *symbol*.

For `address` to return an address, the symbol must be a valid entry in the symbol table. If the `address` method does not find the symbol, it generates a warning and leaves `a` empty.

`scope`

Optionally, you set the scope of the address method. Enter `'local'` or `'global'`. Use `'local'` when the current scope of the program is the desired function scope. If you omit the *`scope`* argument, the `address` method uses `'local'` by default.

Output Arguments

If the `address` method does not find the symbol, it generates a warning and does not return a value for `a`.

The `address` method only returns address information for the first matching symbol in the symbol table.

For Code Composer Studio™

The return value, `a`, is a numeric array with the symbol's address offset, `a(1)`, and page, `a(2)`.

With TI C6000™ processors, the memory page value is 0.

For Eclipse

With Eclipse IDE, the `address` method only returns the symbol address. It does not return a value for page.

The return value, `a`, is the numeric value of the symbol address.

For MULTI®

With MULTI, `address` requires a linker command file (lcf) in your project.

The return value, `a`, is a numeric array with the symbol's address offset, `a(1)`, and page, `a(2)`.

For VisualDSP++®

With VisualDSP++, `address` requires a linker command file (lcf) in your project.

The return value `a` is a numeric array with the symbol's start address, `a(1)`, and memory type, `a(2)`.

Examples

After you load a program to your processor, `address` lets you read and write to specific entries in the symbol table for the program. For example, the following function reads the value of symbol `'ddat'` from the symbol table in the IDE.

```
ddatv = IDE_Obj.read(IDE_Obj.address('ddat'),'double',4)
```

`ddat` is an entry in the current symbol table. `address` searches for the string *`ddat`* and returns a value when it finds a match. `read` returns *`ddat`* to MATLAB software as a double-precision value as specified by the string `'double'`.

To change values in the symbol table, use `address` with `write`:

```
IDE_Obj.write(IDE_Obj.address('ddat'),double([pi 12.3 exp(-1)...
sin(pi/4)]))
```

After executing this write operation, *`ddat`* contains double-precision values for π, 12.3, e^{-1}, and sin(π/4). Use `read` to verify the contents of *`ddat`*:

```
ddatv = IDE_Obj.read(IDE_Obj.address('ddat'),'double',4)
```

MATLAB software returns

```
ddatv =

     3.1416   12.3   0.3679   0.7071
```

See Also

`load` | `read` | `write`

addSourceFiles

Purpose

Add source files to model's build information

Syntax

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

`paths` and *`groups`* are optional.

Arguments

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

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

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

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

  The function removes duplicate source file entries that

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

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

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

`groups` (optional)
: A character array or cell array of character arrays that groups specified source files. You can use groups to

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

You can apply

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

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

Description

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

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

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

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

Examples

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

```
myModelBuildInfo = RTW.BuildInfo;
addSourceFiles(myModelBuildInfo, 'driver.c', ...
'/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 folder to build information `myModelBuildInfo` and place the files in the group `CFiles`.

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

See Also

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

How To

- "Customizing Post-Code-Generation Build Processing"

addSourcePaths

Purpose Add source paths to model's build information

Syntax `addSourcePaths(buildinfo, paths, groups)`

groups is optional.

Arguments

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

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

The function removes duplicate source file entries that

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

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

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

---

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

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

You can apply

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

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

Description

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

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

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

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

Examples

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

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

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

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

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

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

See Also

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

How To

- “Customizing Post-Code-Generation Build Processing”

Purpose

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

Syntax

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

`groups` is optional.

Arguments

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

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

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

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

`groups` (optional)
: A character array or cell array of character arrays that groups specified TMF tokens. You can use groups to

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

  You can apply

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

To...	Specify *groups* as a...
Apply one group name to 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 *tokennames*.

Description

Call the `addTMFTokens` function inside a post code generation command to provide build-time information to help customize makefile generation. The tokens specified in the `addTMFTokens` function call must be handled 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 “Examples” on page 3-43.)

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

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

If You Specify an Optional Argument as a...	The Function...
Character array	Applies the character array to 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 *tokennames*.

Examples

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

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

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

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

How To

- "Customizing Post-Code-Generation Build Processing"

build

Purpose

Build or rebuild current project

Syntax

```
[result,numwarns]=IDE_Obj.build(timeout)
IDE_Obj.build('all')
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`[result,numwarns]=IDE_Obj.build(timeout)` incrementally builds the active project. Incremental builds recompile only source files in your project that you changed or added after the most recent build. `build` uses the file time stamp to determine whether to recompile a file. After recompiling the source files, `build` links the object files to make a new program file.

The value of `result` is 1 when the build process completes successfully. The value of `numwarns` is the number of compilation warnings generated from the build process.

The *`timeout`* argument defines the number of seconds MATLAB waits for the IDE to complete the build process. If the IDE exceeds the timeout period, this method returns a timeout error immediately. The timeout error does not terminate the build process in the IDE. The IDE continues the build process. The timeout error indicates that the build process did not complete before the specified timeout period expired. If you omit the *`timeout`* argument, the `build` method uses a default value of `1000` seconds.

`IDE_Obj.build('all')` rebuilds all the files in the active project.

See Also

`isrunning | open`

Purpose

Close project in IDE window

Syntax

`IDE_Obj.close(filename,'project')`

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

Use `IDE_Obj.close(filename,'project')` to close a specific project, all projects, or the active open project.

For the *filename* argument:

- To close all project files, enter `'all'`.
- To close a specific project, enter the project file name, such as `'myProj'`.If the file is not an open file in the IDE, MATLAB returns a warning message.
- To close the active project, enter `[]`.

With the VisualDSP++ IDE, to close the current project group (if *filename* is `'all'` or `[]`), replace `'project'`with `'projectgroup'`.

Note

- The `open` method no longer supports the `'text'` argument.
- Save changes to your files and projects in the IDE before you use `close`. The `close` method does not save changes, nor does it prompt you to save changes, before it closes the project.

Examples

To close all open project files:

```
IDE_Obj.close('all','project')
```

To close the open project, `myProj`:

```
IDE_Obj.close('myProj','project')
```

To close the active open project:

```
IDE_Obj.close([],'project')
```

With the VisualDSP++ IDE, to close all open project groups:

```
IDE_Obj.close('all','projectgroup')
```

With the VisualDSP++ IDE, to close the active project group:

```
IDE_Obj.close([],'projectgroup')
```

See Also

add | open

Purpose

Files and folders in current IDE window

Syntax

```
IDE_Obj.dir
d=IDE_Obj.dir
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`IDE_Obj.dir` lists the files and folders in the IDE working folder, where `IDE_Obj` is the object that references the IDE. `IDE_Obj` can be either a single object, or a vector of objects. When `IDE_Obj` is a vector, `dir` returns the files and folders referenced by each object.

`d=IDE_Obj.dir` returns the list of files and folders as an M-by-1 structure in `d` with the fields for each file and folder shown in the following table.

Field Name	Description
`name`	Name of the file or folder.
`date`	Date of most recent file or folder modification.
`bytes`	Size of the file in bytes. Folders return 0 for the number of bytes.
`isdirectory`	0 if it is a file, 1 if it is a folder.
`datenum`	The Eclipse IDE and Code Composer Studio IDE also return the modification date as a MATLAB serial date number.

To view the entries in structure `d`, use an index in the syntax at the MATLAB prompt, as shown by the following examples.

- `d(3)` returns the third element in the structure.
- `d(10)` returns the tenth element in the structure `d`.

- `d(4).date` returns the `date` field value for the fourth structure element.

See Also

`open`

Purpose

Properties of IDE handle

Syntax

IDE_Obj.display()

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

IDE_Obj.display() displays the properties and property values of the IDE handleIDE_Obj.

For example, after you creating IDE_Obj with a constructor, using the display method with IDE_Obj returns a set of properties and values:

```
IDE_Obj.display

IDE Object:
  Property1        : valuea
  Property2        : valueb
  Property3     : valuec
  Property4        : valued
```

See Also

get

eclipseide

Purpose

Create handle object to interact with Eclipse IDE

Syntax

```
IDE_Obj = eclipseide
IDE_Obj = eclipseide('timeout', period)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

Before using `eclipseide` for the first time:

- Install the correct software versions of the Eclipse IDE, Eclipse software add-ons, and GNU tools. For detailed information and instructions, see "Working with Eclipse IDE" topic for Eclipse IDE.
- Use the `eclipseidesetup` function to configure and install a plug-in that enables your coder product to interact with Eclipse IDE.

Use *`IDE_Obj`* `= eclipseide` to create an IDE handle object, which you can use to communicate with the Eclipse IDE and processors connected to the Eclipse IDE. After creating the IDE handle object, you can use any of the methods for the Eclipse IDE.

When you use `eclipseide`, your coder product uses the plug-in to open a session with Eclipse. If Eclipse IDE is not already running, the `eclipseide` function starts the Eclipse IDE. The session connects via the IP port number and uses the workspace you specified previously with `eclipseidesetup`.

When you build a model, the software uses `eclipseide` to create an IDE handle object. In that case, the software gets the name of the IDE handle object from the **IDE link handle name** parameter (default value: `IDE_Obj`) in the configuration parameters for the model.

To assign a timeout period to the handle object, enter the following command:

```
IDE_Obj = eclipseide('timeout', period)
```

For *period*, enter the number of seconds that the handle object waits for processor operations (such as `load`) to complete. Operations that exceed the timeout period generate timeout errors. The default period is 10 seconds.

Examples

For example, to create an object handle with a 20-second timeout period, enter:

```
>> IDE_Obj = eclipseide('timeout',20)
Starting Eclipse(TM) IDE...

ECLIPSEIDE Object:
  Default timeout  : 20.00 secs
  Eclipse folder   : C:\eclipse3.4\eclipse
  Eclipse workspace: C:\WINNT\Profiles\rdlugyhe\workspace
  Port number      : 5555
  Processor site   : local
```

See Also

`eclipseidesetup`

eclipseidesetup

Purpose

Configure your coder product to interact with Eclipse IDE

Syntax

```
eclipseidesetup
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

Before using `eclipseidesetup` for the first time, install the correct software versions of the Eclipse IDE, Eclipse software add-ons, and GNU tools. For detailed information and instructions, see "Working with Eclipse IDE" topic for Eclipse IDE.

To avoid potential build errors later on, close Eclipse IDE before you run `eclipseidesetup`. For more information, see Build Errors.

Use `eclipseidesetup` at the MATLAB command line to set up your coder product to interact with Eclipse IDE. This action displays a dialog box which you use to configure and add a plugin to the Eclipse IDE. For detailed instructions and examples, see "Configuring Your MathWorks Software to Work with Eclipse".

When to use `eclipseidesetup`:

- After you install or reinstall the Eclipse IDE.
- Before you use the `eclipseide` constructor function to create an IDE handle object for the first time.

See Also

`eclipseide`

Purpose

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

Syntax

`findIncludeFiles(buildinfo, extPatterns)`

`extPatterns` is optional.

Arguments

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

`extPatterns` (optional)
: A cell array of character arrays that specify patterns of file name extensions for which the function is to search. Each pattern

  - Must start with `*.`
  - Can include any combination of alphanumeric and underscore (_) characters

  The default pattern is `*.h`.

  Examples of valid patterns include

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

Description

The `findIncludeFiles` function

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

Examples

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

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

See Also

`addIncludeFiles` | `getIncludeFiles` | `packNGo`

How To

- "Customizing Post-Code-Generation Build Processing"

Purpose

Halt program execution by processor

Syntax

```
IDE_Obj.halt
IDE_Obj.halt(timeout)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

IDE_Obj.halt stops the program running on the processor. After you issue this command, MATLAB waits for a response from the processor that the processor has stopped. By default, the wait time is 10 seconds. If 10 seconds elapses before the response arrives, MATLAB returns an error. In this syntax, the timeout period defaults to the global timeout period specified in IDE_Obj. Use IDE_Obj.get to determine the global timeout period. However, the processor usually stops in spite of the error message.

To resume processing after you halt the processor, use run. Also, the IDE_Obj.read('pc') function can determine the memory address where the processor stopped after you use halt.

IDE_Obj.halt(timeout) immediately stops program execution by the processor. After the processor stops, halt returns to the host. timeout defines, in seconds, how long the host waits for the processor to stop running. If the processor does not stop within the specified timeout period, the routine returns with a timeout error.

Examples

Use one of the provided demonstration programs to show how halt works. Load and run one of the demonstration projects. At the MATLAB prompt, check whether the program is running on the processor.

```
IDE_Obj.isrunning

ans =

     1
```

```
IDE_Obj.isrunning % Alternate syntax for checking the run status.

ans =

     1
IDE_Obj.halt % Stop the running application on the processor.
IDE_Obj.isrunning

ans =

     0
```

Issuing the halt stops the process on the processor. Checking in the IDE confirms that the process has stopped.

See Also

isrunning | run

Purpose

Insert debug point in file

Syntax

```
IDE_Obj.insert(addr,type,timeout)
IDE_Obj.insert(addr)
IDE_Obj.insert(file,line,type,timeout)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`IDE_Obj.insert(addr,type,timeout)` places a debug point at the provided address of the processor. The IDE_Obj handle defines the processor that will receive the new debug point. The debug point location is defined by `addr`, the desired memory address. The IDEs support several types of debug points. Refer to your IDE help documentation for information on their respective behavior. The following table shows which debug types each IDE supports.

	CCS IDE	Eclipse IDE	MULTI	VisualDSP++
'break' (default)	Yes	Yes	Yes	Yes
'watch'		Yes	Yes	
'probe'	Yes			

The `timeout` parameter defines how long to wait (in seconds) for the insert to complete. If this period is exceeded, the routine returns immediately with a timeout error. In general the action (insert) still occurs, but the timeout value gave insufficient time to verify the completion of the action.

`IDE_Obj.insert(addr)` same as the preceding example, except the `timeout` value defaults to the timeout property specified by the IDE_Obj object. Use `IDE_Obj.get('timeout')` to examine this default timeout value.

`IDE_Obj.insert(file,line,type,timeout)` places a debug point at the specified line in a source file of Eclipse. The FILE parameter gives the name of the source file. LINE defines the line number to receive the breakpoint. Eclipse IDE provides several types of debug points. Refer to the previous list of supported debug point types. Refer to Eclipse IDE documentation for information on their respective behavior.

`IDE_Obj.insert(file,line)` same as the preceding example, except the timeout value defaults to the timeout property specified by the IDE_Obj object. Use `IDE_Obj.get('timeout')` to examine this default timeout value.

See Also

`address` | `run`

Purpose

Determine whether processor is executing process

Syntax

IDE_Obj.isrunning

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

IDE_Obj.isrunning returns 1 when the processor is executing a program. When the processor is halted, isrunning returns 0.

Examples

isrunning lets you determine whether the processor is running. After you load a program to the processor, use isrunning to verify that the program is running.

```
IDE_Obj.load('program.exe','program')
IDE_Obj.run
IDE_Obj.isrunning

ans =

     1
IDE_Obj.halt
IDE_Obj.isrunning

ans =

     0
```

See Also

halt | load | run

getCompileFlags

Purpose

Compiler options from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
: Build information returned by RTW.BuildInfo.

includeGroups (optional)
: A character array or cell array of character arrays that specifies groups of compiler flags you want the function to return.

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

Output Arguments

Compiler options stored in the model's build information.

Description

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

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

Examples

- Get all compiler options stored in build information myModelBuildInfo.

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

```
compflags =

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

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

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

compflags =

    '-Zi -Wall'
```

- Get 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

`addCompileFlags` | `getDefines` | `getLinkFlags`

How To

- "Customizing Post-Code-Generation Build Processing"

getDefines

Purpose

Preprocessor macro definitions from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

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

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

Output Arguments

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

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

Description

The `getDefines` function returns preprocessor macro definitions stored in the model'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'}, 'OPTS');
[defs names values]=getDefines(myModelBuildInfo);
defs

defs =

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

names

names =

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

```
values

values =

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

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

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

defs =

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

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

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

defs =

    '-DPRODUCTION'
```

See Also

`addDefines` | `getCompileFlags` | `getLinkFlags`

How To

- "Customizing Post-Code-Generation Build Processing"

getFullFileList

Purpose

All files from model's build information

Syntax

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

`fcase` is optional.

Input Arguments

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

`fcase` (optional)
: The string `'source'`, `'include'`, or `'nonbuild'`. If the argument is omitted, the function returns 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.

Output Arguments

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

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

Description

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

How To

- "Customizing Post-Code-Generation Build Processing"

getIncludeFiles

Purpose

Include files from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

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

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

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

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

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

`includeGroups` (optional)
: A character array or cell array of character arrays that specifies groups of include files you want the function to return.

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

Output Arguments

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

Description

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

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

Examples

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

```
myModelBuildInfo = RTW.BuildInfo;
addIncludeFiles(myModelBuildInfo, {'etc.h' 'etc_private.h'...
'mytypes.h'}, {'/etc/proj/etclib' '/etcproj/etc/etc_build'...
```

```
'/common/lib'}, {'etc' 'etc' 'shared'});
incfiles=getIncludeFiles(myModelBuildInfo, true, false);
incfiles

incfiles =

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

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

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

incfiles =

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

See Also

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

How To

- "Customizing Post-Code-Generation Build Processing"

Purpose

Include paths from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

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

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

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

includeGroups (optional)
: A character array or cell array of character arrays that specifies groups of include paths you want the function to return.

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

Output Arguments

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

Description

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

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

Examples

- Get all include paths stored in build information myModelBuildInfo.

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

incpaths =

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

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

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

incpaths =

    '\common\lib''
```

See Also

addIncludePaths | getIncludeFiles | getSourceFiles | getSourcePaths

How To

- “Customizing Post-Code-Generation Build Processing”

Purpose

Link options from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

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

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

Output Arguments

Linker options stored in the model's build information.

Description

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

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

Examples

- Get all linker options stored in build information myModelBuildInfo.

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

linkflags =

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

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

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

linkflags =

    '-MD -Gy'
```

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

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

linkflags =

    '-T'
```

See Also

addLinkFlags | getCompileFlags | getDefines

How To

- “Customizing Post-Code-Generation Build Processing”

getNonBuildFiles

Purpose Nonbuild-related files from model's build information

Syntax

```
files=getNonBuildFiles(buildinfo, concatenatePaths,
replaceMatlabroot, includeGroups, excludeGroups)
```

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

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

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

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

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

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

includeGroups (optional)
: A character array or cell array of character arrays that specifies groups of nonbuild files you want the function to return.

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

Output Arguments

Names of nonbuild files stored in the model'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`

How To

- "Customizing Post-Code-Generation Build Processing"

Purpose

Source files from model's build information

Syntax

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

includeGroups and *excludeGroups* are optional.

Input Arguments

buildinfo
Build information returned by `RTW.BuildInfo`.

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

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

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

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

If You Specify...	The Function...
`true`	Replaces path tokens, such as `$(MATLAB_ROOT)` and `$(START_DIR)`, with the absolute path string.
`false`	Does not replace path tokens with the absolute path string.

includeGroups (optional)
: A character array or cell array of character arrays that specifies groups of source files you want the function to return.

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

Output Arguments

Names of source files stored in the model'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 expansions of path tokens in the output it returns. Using optional *includeGroups* and *excludeGroups* arguments, you can selectively include or exclude groups of source files the function returns.

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

Examples

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

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

```
srcfiles

srcfiles =

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

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

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

incfiles =

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

See Also

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

How To

- "Customizing Post-Code-Generation Build Processing"

getSourcePaths

Purpose

Source paths from model's build information

Syntax

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

`includeGroups` and *`excludeGroups`* are optional.

Input Arguments

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

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

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

`includeGroups` (optional)
: A character array or cell array of character arrays that specifies groups of source paths you want the function to return.

`excludeGroups` (optional)
: A character array or cell array of character arrays that specifies groups of source paths you do not want the function to return.

Output Arguments

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

Description

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

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

Examples

- Get all source paths stored in build information myModelBuildInfo.

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

srcpaths =

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

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

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

srcpaths =

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

- Get a path stored in build information myModelBuildInfo. First get the path without replacing $(MATLAB_ROOT) with an absolute path, then get it with replacement. The MATLAB root folder in this case is \\myserver\myworkspace\matlab.

```
myModelBuildInfo = RTW.BuildInfo;
addSourcePaths(myModelBuildInfo, fullfile(matlabroot,...
 'rtw', 'c', 'src'));
```

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

ans =

$(MATLAB_ROOT)\rtw\c\src

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

ans =

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

See Also

addSourcePaths | getIncludeFiles | getIncludePaths | getSourceFiles

How To

- “Customizing Post-Code-Generation Build Processing”

Purpose

Load program file onto processor

Syntax

`IDE_Obj.load(filename,timeout)`

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`IDE_Obj.load(filename,timeout)` loads the file specified by the *`filename`* argument to the processor.

The *`filename`* argument can include a full path to the file, or the name of a file in the IDE working folder.

With the VisualDSP++, MULTI, and Code Composer Studio IDEs, you can use the cd method to check or modify the IDE working folder.

For MULTI, you can add an *`option`* argument after *`filename`* to specify options for the `'prepare_target'` command in MULTI debugger. Refer to the MULTI documentation for information on `'prepare_target'`.

Only use `load` with program files created by the IDE build process.

The *`timeout`* argument defines the number of seconds MATLAB waits for the load process to complete. If the time-out period expires before the load process returns a completion message, MATLAB generates an error and returns. Usually the program load process works correctly in spite of the error message.

If you omit the *`timeout`* argument, `load` uses the timeout property of the IDE handle object, which you can get by entering `IDE_Obj.get('timeout')`.

Using load with Eclipse IDE

With Eclipse IDE:

- Before using `load`, use `activate` to make the project associated with the executable file active.

- For the *filename* argument, use a relative or absolute path to specify the executable file.

A relative path consists of:

project/configuration/executablefile

An absolute path consists of:

workspace/project/configuration/executablefile

If the *workspace* is not the active workspace when you use `load`, the software generates errors.

If the *project* is not the active project when you use `load`, the software makes the project active.

If the software generates socket server errors when you use methods with a Eclipse IDE handle object, such as `IDE_Obj`:

1. Delete the handle object from the MATLAB workspace.
2. Reconnect to the Eclipse IDE using the `eclipseide` constructor.

Examples

```
IDE_Obj.load(programfile)
run(id)
```

See Also

`dir` | `open`

Purpose

Create project, library, or build configuration in IDE

Syntax

`IDE_Obj.new('name','type')`

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`IDE_Obj.new('name','type')` creates a project, library, or build configuration in the IDE.

The `name` argument specifies the name of the new project, library, or build configuration

The type argument specifies whether to create a project, library, or build configuration. The options are:

- `'project'` — Executable project. Sometimes this file is called a "DSP executable file".
- `'projlib'` — Library project.
- `'projext'` — External make project. Only the CCS IDE supports this option.
- `'buildcfg'` — Build configuration in the active project. Only the VisualDSP++ and CCS IDEs support this option.

When *type* is `'project'` or `'projlib'` , *name* can include the full path to the new file. You can use the path to differentiate two files with the same name. If you omit the path, the `new` method creates the file or project in the current IDE working folder.

If you omit the *type* argument, and the *name* argument does not include a file extension, *type* defaults to `'project'`.

When *type* is `'buildcfg'`, use a unique name to differentiate the build configuration from other build configurations in the active project.

The `new` method no longer supports `'text'` as a *type* argument.

Examples

```
IDE_Obj.new('my_project','project') #Create an IDE project, 'my_project.gpj'
IDE_Obj.new('my_build_config','buildcfg') #Create a build configuration.
```

See Also

activate | close

Purpose

Open project in IDE

Syntax

```
IDE_Obj.open(filename,filetype,timeout)
IDE_Obj.open(myproject)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

IDE_Obj.open(*filename*,*filetype*,*timeout*) opens a project in the IDE.

Use the *filename* argument to specify the file name, including the file name extension. If the *filename* does not include a file name extension, you can specify the file type using the *filetype* argument. If the file does not exist in the current project or folder path, MATLAB returns a warning and returns control to MATLAB.

For the optional *filetype* argument, you can specify the following types.

	CCS IDE	Eclipse IDE	MULTI IDE	VisualDSP++ IDE
'project' — Project files	Yes	Yes	Yes	Yes
'ProjectGroup' — Project group files	No	No	No	Yes
'program' — Target program file (executable)	No. Use load instead.	No	Yes	No

If you omit the *filetype* argument, *filetype* defaults to 'project'.

The optional *timeout* argument determines the number of seconds MATLAB waits for the IDE to finish opening the file before returning an error. If you omit the *timeout* argument, the open method uses the timeout property of the IDE handle object (IDE_Obj) instead. The timeout error does not terminate the loading process on the IDE.

Note The `open` method no longer supports the `'text'`, `'program'`, or `'workspace'` arguments.

Examples

`IDE_Obj.open(myproject)` opens the myproject project in the IDE.

See Also

`| dir | load | new`

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 folder	`'packType'`	`'flat'` (default)
Package model code files hierarchically in a primary zip file that contains three secondary zip files: • `mlrFiles.zip` — files in your *`matlabroot`* folder tree • `sDirFiles.zip` — files in and under your build folder • `otherFiles.zip` — required files not in the *`matlabroot`* or `start` folder trees	`'packType'`	`'hierarchical'`Paths for files in the secondary zip files are relative to the root folder of the primary zip file.
Specify a file name for the primary zip file	`'fileName'`	`'`*`string`*`'` Default:`'`*`model`*`.zip'` If you omit the `.zip` file extension, the function adds it for you.

Description

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

- Source files (for example, `.c` and `.cpp` 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 folder structure in a zip file named *`model`*`.zip`. You can tailor the output by specifying property name and value pairs as explained above.

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

Note The `packNGo` function potentially can modify the build information passed in the first `packNGo` argument. For example, as part of packaging model code, `packNGo` finds include files from all source and include paths recorded in the model's build information and adds them to the build information.

Examples

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

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

  Then, rebuild the model.

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

  ```
  cd zingbat_grt_rtw;
  load buildInfo.mat
  ```

```
packNGo(buildInfo, {'packType', 'hierarchical', ...
 'fileName', 'portzingbit'});
```

How To

- “Customizing Post-Code-Generation Build Processing”
- “Relocating Code to Another Development Environment”
- “packNGo Function Limitations”

pwd

Purpose

Working folder used by Eclipse

Syntax

`wd= IDE_Obj.pwd`

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

Use `wd= IDE_Obj.pwd` to get the working folder of the Eclipse IDE. This value is the same as the Eclipse IDE workspace folder.

Examples

To get the Eclipse IDE working folder:

```
IDE_Obj = eclipseide;
wd = IDE_Obj.pwd

wd =

C:\WINNT\Profiles\rdlugyhe\workspace
```

See Also

`dir`

Purpose

Read data from processor memory

Syntax

```
mem=IDE_Obj.read(address)
mem=IDE_Obj.read(...,datatype)
mem=IDE_Obj.read(...,count)
mem=IDE_Obj.read(...,memorytype)
mem=IDE_Obj.read(...,timeout)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`mem=IDE_Obj.read(address)` returns a block of data values from the memory space of the processor referenced by `IDE_Obj`. The block to read begins from the DSP memory location given by the *`address`* argument. The data is read starting from *`address`* without regard to type-alignment boundaries in the processor. Conversely, the byte ordering defined by the data type is automatically applied.

The *`address`* argument is a decimal or hexadecimal representation of a memory address in the processor. In all cases, the full memory address consist of two parts:

- The start address
- The memory type

You can define the memory type value can be explicitly using a numeric vector representation of the address.

Alternatively, the `IDE_Obj` object has a default memory type value that is applied if the memory type value is not explicitly incorporated in the passed address parameter. In DSP processors with only a single memory type, it is possible to specify all addresses using the abbreviated (implied memory type) format by setting the `IDE_Obj` object memory type value to zero.

Note You cannot read data from processor memory while the processor is running.

Provide the *address* argument either as a numerical value that is a decimal representation of the DSP memory address, or as a string that `read` converts to the decimal representation of the start address. (Refer to function `hex2dec` in the *MATLAB Function Reference*. `read` uses `hex2dec` to convert the hexadecimal string to a decimal value).

The examples in the following table demonstrate how `read` uses the `address` parameter.

address Parameter Value	Description
`131082`	Decimal address specification. The memory start address is 131082 and memory type is 0. This action is the same as specifying [131082 0].
`[131082 1]`	Decimal address specification. The memory start address is 131082 and memory type is 1.
`'2000A'`	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify `address` as a cell array. You can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1}=131072;
myadddress1{2}='Program(PM) Memory';
```

```
myaddress2 myaddress2{1}='20000';
myadddress2{2}='Program(PM) Memory';
```

`myaddress3 myaddress3{1}=131072; myaddress3{2}=0;`

`mem=`*`IDE_Obj`*`.read(...,`*`datatype`*`)` where the input argument `datatype` defines the interpretation of the raw values read from DSP memory. Parameter `datatype` specifies the data format of the raw memory image. The data is read starting from `address` without regard to data type alignment boundaries in the processor. The byte ordering defined by the data type is automatically applied. This syntax supports the following MATLAB data types.

MATLAB Data Type	Description
`double`	IEEE double-precision floating point value
`single`	IEEE single-precision floating point value
`uint8`	8-bit unsigned binary integer value
`uint16`	16-bit unsigned binary integer value
`uint32`	32-bit unsigned binary integer value
`int8`	8-bit signed two's complement integer value
`int16`	16-bit signed two's complement integer value
`int32`	32-bit signed two's complement integer value

The `read` method does not coerce data type alignment. Some combinations of `address` and `datatype` will be difficult for the processor to use.

`mem=`*`IDE_Obj`*`.read(...,`*`count`*`)` adds the `count` input parameter that defines the dimensions of the returned data block `mem`. To read a block

of multiple data values. Specify count to determine how many values to read from address. count can be a scalar value that causes read to return a column vector that has count values. You can perform multidimensional reads by passing a vector for count. The elements in the input vector of count define the dimensions of the returned data matrix. The memory is read in column-major order. count defines the dimensions of the returned data array mem as shown in the following table.

- n — Read n values into a column vector.
- [m,n] — Read m-by-n values into m by n matrix in column-major order.
- [m,n,...] — Read a multidimensional matrix m-by-n-by…of values into an m-by-n-by…array.

To read a block of multiple data values, specify the input argument count that determines how many values to read from address.

mem=*IDE_Obj*.read(…,*memorytype*) adds an optional input argument memorytype. Object IDE_Obj has a default memory type value 0 that read applies if the memory type value is not explicitly incorporated into the passed address parameter.

In processors with only a single memory type, it is possible to specify all addresses using the implied memory type format by setting the IDE_Objmemorytype property value to zero.

Using read with MULTI

Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
'program(pm) memory'	0	Blackfin and SHARC
'data(dm) memory'	1	SHARC

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
'data(dm) short word memory'	2	SHARC
'external data(dm) byte memory'	3	SHARC
'boot(prom) memory'	4	SHARC

mem=*IDE_Obj*.read(...,*timeout*) adds the optional parameter *timeout* that defines how long, in seconds, MATLAB waits for the specified read process to complete. If the time-out period expires before the read process returns a completion message, MATLAB returns an error and returns. Usually the read process works correctly in spite of the error message.

Examples

This example reads one 16-bit integer from memory on the processor.

```
mlvar = IDE_Obj.read(131072,'int16')
```

131072 is the decimal address of the data to read.

You can read more than one value at a time. This read command returns 100 32-bit integers from the address 0x20000 and plots the result in MATLAB.

```
data = IDE_Obj.read('20000','int32',100)
plot(double(data))
```

See Also

write

reload

Purpose

Reload most recent program file to processor signal processor

Syntax

```
s = IDE_Obj.reload(timeout)
s = IDE_Obj.reload
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`s = IDE_Obj.reload(timeout)` resends the most recently loaded program file to the processor. If you have not loaded a program file in the current session (so there is no previously loaded file), `reload` returns the null entry `[]` in `s` indicating that it could not load a file to the processor. Otherwise, `s` contains the full path name to the program file. After you reset your processor or after any event produces changes in your processor memory, use `reload` to restore the program file to the processor for execution.

To limit the time the IDE spends trying to reload the program file to the processor, *`timeout`* specifies how long the load process can take. If the load process exceeds the timeout limit, the IDE stops trying to load the program file and returns an error stating that the time period expired. Exceeding the allotted time for the reload operation usually indicates that the reload was successful but the IDE did not receive confirmation before the timeout period passed.

`s = IDE_Obj.reload` reloads the most recent program file, using the *`timeout`* value set when you created link `IDE_Obj`, the global timeout setting.

Using reload with Multiprocessor Boards

When your board contains more than one processor, `reload` calls the reloading function for each processor represented by `IDE_Obj`, reloading the most recently loaded program on each processor.

This action is the same as calling `reload` for each processor individually through IDE handle objects for each one.

Examples

After you create an object that connects to the IDE, use the available methods to reload your most recently loaded project. If you have not loaded a project in this session, `reload` returns an error and an empty value for `s`. Loading a project eliminates the error. First, create an IDE handle object, such as IDE_Obj, using the constructor for your IDE.

```
s=IDE_Obj.reload(23)
Warning: No action taken - load a valid Program file before
you reload...

s =

     ''

IDE_Obj.open('D:\ti\tutorial\sim62xx\gelsolid\hellodsp.pjt','project')

IDE_Obj.build

IDE_Obj.load('hellodsp.pjt') #This file extension varies by IDE
IDE_Obj.halt
s=IDE_Obj.reload(23)

s =

D:\ti\tutorial\sim62xx\gelsolid\Debug\hellodsp.out
```

See Also

| `load` | `open`

Purpose

Remove file, project, or breakpoint

Syntax

```
IDE_Obj.remove(filename,filetype)
IDE_Obj.remove(addr,debugtype,timeout)
IDE_Obj.remove(filename,line,debugtype,timeout)
IDE_Obj.remove(all,break)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`IDE_Obj.remove(filename,filetype)` deletes a file from the active project in the IDE or deletes the project.

`IDE_Obj.remove(addr,debugtype,timeout)` removes a debug point from an address in the program.

`IDE_Obj.remove(filename,line,debugtype,timeout)` removes a debug point from a line in a source file.

`IDE_Obj.remove(all,break)` removes all of the breakpoints and waits for completion.

Input Arguments

`IDE_Obj`

Enter the name of the IDE link handle for your IDE. Create an IDE link handle before you use the `remove` method. .

`filename`

Replace *`filename`* with the name of the file you are removing, or the source file from which you are removing debug points. If the file is not located in the active project, MATLAB returns a warning instead of completing the action.

`filetype`

To remove a project, enter `'project'`. To remove a source file, enter `'text'`.

Default: `'text'`

`addr`

Enter the memory address of the debug point. Enter `'all'` to remove all of the breakpoints.

`debugtype`

Enter the type of debug point to remove. The IDE provide several types of debug points. Refer to the IDE help documentation for information on their respective behavior.

Default: `'break'` (breakpoint)

`line`

Enter the line number of the debug point located in a file.

`timeout`

Enter a time limit, in seconds, for the method to complete an action.

Examples

After you have a project in the IDE, you can delete files from it using `remove` from the MATLAB software command line. For example, build a project and load the resulting `.out` file. With the project build complete, load your `.out` file by typing

```
IDE_Obj.load('filename.out')
```

Now remove one file from your project

```
IDE_Obj.remove('filename')
```

You see in the IDE that the file no longer appears.

See Also

`add | | open`

restart

Purpose

Reload most recent program file to processor signal processor

Syntax

```
IDE_Obj.restart
IDE_Obj.restart(timeout)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

IDE_Obj.restart issues a restart command in the IDE debugger. The behavior of the restart process depends on the processor. Refer to the documentation for your IDE for details about using restart with various processors.

When IDE_Obj is an array that contains more than one processor, each processor calls restart in sequence.

IDE_Obj.restart(timeout) adds the optional timeout input argument. timeout defines an upper limit in seconds on the period the restart routine waits for completion of the restart process. If the time-out period is exceeded, restart returns control to MATLAB with a time-out error. In general, restart causes the processor to initiate a restart, even if the time-out period expires. The time-out error indicates that the restart confirmation was not received before the time-out period elapsed.

See Also

halt | isrunning | run

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

The `parameters` substructure includes the following fields:

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

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

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

Examples

Return global parameter structure for model rtwdemo_rsimtf to param_struct:

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

param_struct =

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

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”

rsimsetrtpparam

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.

Input Arguments

`rtP`

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

`idx`

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

`paramValue`

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

`paramName`

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

Output Arguments

`rtP`

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

Definitions

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

Examples

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

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

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

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

See Also

`rsimgetrtp`

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.

Input Arguments

`model`

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

`build_spec`

Structure of field and value pairs that define a build specification; 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

Field	Value
	library rules only if the conversion needs the libraries.
libSuffix (optional)	String that specifies the suffix, including the file type extension, to append to the name of each library (for example, .a or _vc.lib). The string must include a period (.). Set the suffix with either this field or the TargetLibSuffix parameter. If you specify a suffix with both mechanisms, the TargetLibSuffix setting overrides the setting of this field.
intOnlyBuild (optional)	Boolean flag. When set to true, indicates the function optimizes the libraries so that they compile from integer code only. Applies to ERT-based targets only.
makeOpts (optional)	String that specifies an option to include in the rtwMake command line.
addLibs (optional)	Cell array of structures that specify the libraries to build that an rtwmakecfg function does not specify. Define each structure with two fields that are character arrays: • libName — name of the library without a suffix • libLoc — location for the precompiled library The TMF can specify other libraries and how to build them. Use this field if you must precompile libraries.

Examples

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

```
% Specify the library suffix
```

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

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

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

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

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')
blockHandle = rtwbuild('subsystem', 'Mode',
   'ExportFunctionCalls',
   'ExportFunctionInitializeFunctionName', 'fcnname')
```

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 **Code Generation** 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 Simulation Diagnostics 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 Embedded Coder software and set **Create block** to `SIL` in the **Code Generation > SIL and PIL Verification** pane of the Configuration Parameters dialog box, `rtwbuild` returns a nonempty block handle, *`blockHandle`*, to an automatically generated 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 Embedded Coder software to export function-call subsystems.

`blockHandle` `= rtwbuild('`*`subsystem`*`', 'Mode', 'ExportFunctionCalls', 'ExportFunctionInitializeFunctionName', '`*`fcnname`*`')` initiates the build process to export function calls from the specified subsystem and specifies *`fcnname`* as the name of the initialize function of your exported functions. You must be licensed for 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 **Code Generation** pane of the Configuration Parameters dialog box and click **Build**.
- Press **Ctrl+B**.
- Select **Tools > Code Generation > 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
- "Program Building, Interaction, and Debugging"

- Automatic S-Function Wrapper Generation
- Exporting Function-Call Subsystems

RTW.getBuildDir

Purpose

Build folder information for specified model

Syntax

`struct=RTW.getBuildDir(modelName)`

Input Arguments

modelName

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

Output Arguments

Structure containing the following build folder information about the specified model:

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

Description

The `RTW.getBuildDir` function returns build folder 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 folder 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.

Examples

Return build folder information for the model `mymmodel`.

```
>> 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 folder is the build folder of the model.

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

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

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

Input Arguments

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

Examples

Rebuild the `rtwdemo_f14` model:

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

Rebuild the model in a specified path:

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

How To

- “Rebuilding a Model”

Purpose

Generate report documenting generated code for model

Syntax

```
rtwreport(model)
rtwreport(model, folder)
```

Description

`rtwreport(model)` generates a report that shows the generated code for a model. *`model`* 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:

- 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 folder

By default, the Simulink Coder software names the generated report `codegen.html` and places the file in your current folder.

`rtwreport(model, folder)` includes a user-specified folder. *`folder`* is a string enclosed in quotes specifying the name of a folder. The Simulink Coder software places the generated report in the parent folder of the folder you specify. The Simulink Coder project folder (`slprj`) must be in the parent folder. If the user-specified folder cannot be found, an error results and the Simulink Coder software does not generate code.

Examples

Generate a report for model `rtwdemo_codegenrpt`:

```
rtwreport('rtwdemo_codegenrpt');
```

Alternatives

In the model window, select **Tools > Report Generator**. See *Simulink® Report Generator™ User's Guide* for more information.

Tutorials

- "Simulink Report Generator Report"

How To

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

Purpose

Trace block to generated code

Syntax

`rtwtrace(`*`blockpath`*`)`

Description

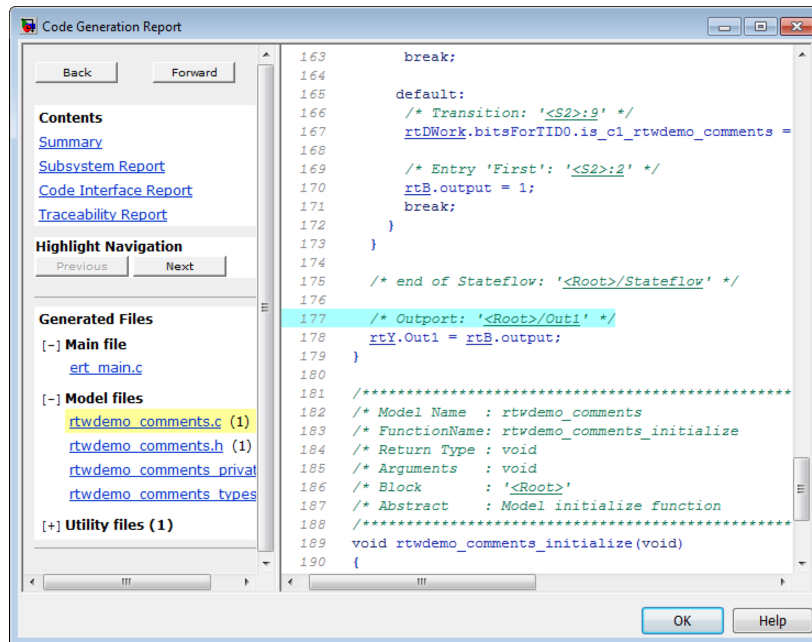
`rtwtrace(`*`blockpath`*`)` opens an HTML code generation report, that displays contents of the source code file, and highlights the line of code corresponding to the specified block. *`blockpath`* is a string enclosed in quotes specifying the full Simulink block path, for example, `'`*`model_name/block_name`*`'`.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

Examples

After enabling model to code navigation and generating code for the demo model `rtwdemo_comments`, use the following command to trace to the source code for block `Out1` in the model:

```
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 **Code Generation > Navigate to code**.

How To

- "Tracing Model Objects to Generated Code"
- "Model-to-code" on page 6-48

Purpose

Execute program loaded on processor

Syntax

```
IDE_Obj.run
IDE_Obj.run('runopt')
IDE_Obj.run(...,timeout)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`IDE_Obj.run` runs the program file loaded on the referenced processor, returning immediately after the processor starts running. Program execution starts from the location of program counter (PC). Usually, the PC is positioned at the top of the executable file. However, if you stopped a running program with `halt`, the PC may be anywhere in the program. `run` starts the program from the PC current location.

If `IDE_Obj` references more the one processor, each processors calls `run` in sequence.

`IDE_Obj.run('runopt')` includes the parameter `runopt` that defines the action of the `run` method. The options for `runopt` are listed in the following table.

runopt string	Description
'run'	Executes the run and waits to confirm that the processor is running, and then returns to MATLAB.
'runtohalt'	Executes the run but then waits until the processor halts before returning. The halt can be the result of the PC reaching a breakpoint, or by direct interaction with the IDE, or by the normal program exit process.

runopt string	Description
'tohalt'	Waits until the running program has halted. Unlike the other options, this selection does not execute a run, it simply waits for the running program to halt.
'main'	This option resets the program and executes a run until the start of function 'main'.
'tofunc'	This option must be followed by an extra parameter *funname*, the name of the function to run to: `IDE_Obj.run('tofunc',funcname)` This executes a run from the present PC location until the start of function *funcname* is reached. If *funcname* is not along the program's normal execution path, *funcname* is not reached and the method times out.

In the 'run' and 'runtohalt' cases, a halt can be caused by a breakpoint, a direct interaction with the IDE, or by a normal program exit.

The following table shows the availability of the *runopt* options by IDE.

	CCS IDE	Eclipse IDE	MULTI IDE	VisualDSP++ IDE
'run'	Yes	Yes	Yes	Yes
'runtohalt'	Yes	Yes	Yes	Yes
'tohalt'	Yes		Yes	
'main'	Yes		Yes	
'tofunc'	Yes		Yes	

IDE_Obj.run(...,*timeout*) adds input argument timeout, to allow you to set the time out to a value different from the global timeout value. The *timeout* value specifies how long, in seconds, MATLAB waits for the processor to start executing the loaded program before returning.

Most often, the 'run' and 'runtohalt' options cause the processor to initiate execution, even when a timeout is reached. The timeout indicates that the confirmation was not received before the timeout period elapsed.

See Also

halt | load |

Simulink.fileGenControl

Purpose

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

Syntax

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

Description

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

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

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

`Simulink.fileGenControl('setConfig', 'config', `*`cfg`*`, 'keepPreviousPath', true, 'createDir', true)` sets the file generation control configuration for the current MATLAB session by passing a handle to an instance of the `Simulink.FileGenConfig`

object containing values for the `CacheFolder` and/or `CodeGenFolder` parameters. To keep the previous values of `CacheFolder` and `CodeGenFolder` in the MATLAB path, specify `'keepPreviousPath'` with the value `true`. To create the specified file generation folders if they do not already exist, specify `'createDir'` with the value `true`.

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

Input Arguments

`action`

String specifying one of the following actions:

Action	Description
`getConfig`	Returns a handle to an instance of the `Simulink.FileGenConfig` object containing the current values of the `CacheFolder` and `CodeGenFolder` parameters.
`reset`	Reinitializes the `CacheFolder` and `CodeGenFolder` parameters to the values provided by the Simulink preferences “Simulation cache folder” and “Code generation folder”.

Action	Description
set	Sets the CacheFolder and/or CodeGenFolder parameters for the current MATLAB session by directly passing values.
setConfig	Sets the CacheFolder and/or CodeGenFolder parameters for the current MATLAB session by passing a handle to an instance of the Simulink.FileGenConfig object.

'config', cfg

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

'CacheFolder', cacheFolderPath

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

'CodeGenFolder', codeGenFolderPath

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

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

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

`'keepPreviousPath', true`

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

`'createDir', true`

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

Output Arguments

`cfg`

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

Examples

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

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

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

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

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

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

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

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

Alternatives

Instead of setting the `CacheFolder` and `CodeGenFolder` parameters just for the current MATLAB session, you can set the Simulink preferences "Simulation cache folder" and "Code generation folder", which provide the initial MATLAB session defaults. The preferences can be set using the Simulink Preferences dialog box or using the MATLAB command `set_param`.

See Also

"Simulation cache folder" | "Code generation folder"

How To

•

switchTarget

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.

Input Arguments

`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 MATLAB language 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';
% 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 **Code Generation > 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

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

Syntax

`tlc [-options] [file]`

Description

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

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

`tlc [-options] [file]`

You can change the default behavior by specifying one or more compilation *options* as described in "Options" on page 3-134

Options

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

- "-r Specify Simulink® Coder filename" on page 3-135
- "-v Specify verbose level" on page 3-135
- "-l Specify path to local include files" on page 3-135
- "-m Specify maximum number of errors" on page 3-135
- "-O Specify the output file path" on page 3-135
- "-d[a|c|n|o] Invoke debug mode" on page 3-135

- "-a Specify parameters" on page 3-136
- "-p Print progress" on page 3-136
- "-lint Performance checks and runtime statistics" on page 3-136
- "-xO Parse only" on page 3-136

-r Specify Simulink Coder filename

`-r` *file_name*

Specify the filename that you want to translate.

-v Specify verbose level

`-v` *number*

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

-l Specify path to local include files

`-l` *path*

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

-m Specify maximum number of errors

`-m` *number*

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

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

-O Specify the output file path

`-O` *path*

Specify the folder `path` to place output files.

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

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

`-da` execute any %assert directives

`-dc` invoke the TLC command line debugger

`-dn` produce log files, which indicate those lines hit and those lines missed during compilation.

`-do` disable debugging behavior

-a Specify parameters

`-a` *identifier* = *expression*

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

-p Print progress

`-p` *number*

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

-lint Performance checks and runtime statistics

`-lint`

Perform simple performance checks and collect runtime statistics.

-xO Parse only

`-xO`

Parse only a TLC file; do not execute it.

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 folder 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

---

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

Examples

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

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

before =

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

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

ans =

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


ans =

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


ans =

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

See Also

addIncludeFiles | addIncludePaths | addSourceFiles | addSourcePaths | updateFileSeparator

How To

- “Customizing Post-Code-Generation Build Processing”

updateFileSeparator

Purpose

Change file separator used in model's build information

Syntax

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

Arguments

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

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

Description

The `updateFileSeparator` function changes all instances of the current file separator (/ or \) in a model's build information to the specified file separator.

The default value for the file separator matches the value returned by the MATLAB command `filesep`. For makefile based builds, you can override the default by defining a separator with the `MAKEFILE_FILESEP` macro in the template makefile (see "Cross-Compiling Code Generated on a Microsoft® Windows System". If the `GenerateMakefile` parameter is set, the Simulink Coder software overrides the default separator and updates the model's build information after evaluating the `PostCodeGenCommand` configuration parameter.

Examples

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

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

See Also

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

How To

- "Customizing Post-Code-Generation Build Processing"
- "Cross-Compiling Code Generated on a Microsoft Windows System"

Purpose

Write data to processor memory block

Syntax

```
mem=IDE_Obj.write(address,data)
mem=write(...,datatype)
mem=IDE_Obj.write(...,memorytype)
mem=IDE_Obj.write(...,timeout)
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

`mem=IDE_Obj.write(address,data)` writes `data`, a collection of values, to the memory space of the DSP processor referenced by `IDE_Obj`.

The *`data`* argument is a scalar, vector, or array of values to write to the memory of the processor. The block to write begins from the DSP memory location given by the input parameter `address`.

The method writes the data starting from *`address`* without regard to type-alignment boundaries in the DSP. Conversely, the byte ordering of the data type is automatically applied.

Note You cannot write data to processor memory while the processor is running.

The *`address`* argument is a decimal or hexadecimal representation of a memory address in the processor. In all cases, the full memory address consist of two parts: the start address and the memory type. The memory type value can be explicitly defined using a numeric vector representation of the address.

Alternatively, the `IDE_Obj` object has a default memory type value which is applied if the memory type value is not explicitly incorporated into the passed address parameter. In DSP processors with only a single memory type, by setting the `IDE_Obj` object memory type value to

zero it is possible to specify all addresses using the abbreviated (implied memory type) format.

You provide the *address* argument either as a numerical value that is a decimal representation of the DSP memory address, or as a string that `write` converts to the decimal representation of the start address. (Refer to function `hex2dec` in the *MATLAB Function Reference* that `read` uses to convert the hexadecimal string to a decimal value).

The following examples demonstrate how `write` uses the *address* argument.

address Parameter Value	Description
`131082`	Decimal address specification. The memory start address is 131082 and memory type is 0. This action is the same as specifying [131082 0].
`[131082 1]`	Decimal address specification. The memory start address is 131082 and memory type is 1.
`'2000A'`	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify *address* as cell array, in which case you can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1} = 131072; myadddress1{2} =
'Program(PM) Memory';
```

```
myaddress2 myaddress2{1} = '20000'; myadddress2{2} =
'Program(PM) Memory';
```

```
myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;
```

`mem=write(...,`*`datatype`*`)` where the *`datatype`* argument defines the interpretation of the raw values written to DSP memory. The *`datatype`* argument specifies the data format of the raw memory image. The data is written starting from `address` without regard to data type alignment boundaries in the DSP. The byte ordering of the data type is automatically applied. The following MATLAB data types are supported.

MATLAB Data Type	Description
`double`	IEEE double-precision floating point value
`single`	IEEE single-precision floating point value
`uint8`	8-bit unsigned binary integer value
`uint16`	16-bit unsigned binary integer value
`uint32`	32-bit unsigned binary integer value
`int8`	8-bit signed two's complement integer value
`int16`	16-bit signed two's complement integer value
`int32`	32-bit signed two's complement integer value

`write` does not coerce data type alignment. Some combinations of *`address`* and *`datatype`* will be difficult for the processor to use.

`mem=`*`IDE_Obj`*`.write(...,`*`memorytype`*`)` adds an optional *`memorytype`* argument. Object `IDE_Obj` has a default memory type value 0 that `write` applies if the memory type value is not explicitly incorporated into the passed address parameter. In processors with only a single memory type, it is possible to specify all addresses using the implied

memory type format by setting the value of the `IDE_Obj` memorytype property to zero.

`mem=IDE_Obj.write(...,timeout)` adds the optional *timeout* argument, which the number of seconds MATLAB waits for the write process to complete. If the *timeout* period expires before the write process returns a completion message, MATLAB throws an error and returns. Usually the process works correctly in spite of the error message.

Using write with VisualDSP++ IDE

Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
'program(pm) memory'	0	Blackfin and SHARC
'data(dm) memory'	1	SHARC
'data(dm) short word memory'	2	SHARC
'external data(dm) byte memory'	3	SHARC
'boot(prom) memory'	4	SHARC

Examples

Example with VisualDSP++ IDE

These three syntax examples demonstrate how to use `write` in some common ways. In the first example, write an array of 16-bit integers to location [131072 1].

```
IDE_Obj.write([131072 1],int16([1:100]));
```

Now write a single-precision IEEE floating point value (32-bits) at address 2000A(Hex).

```
IDE_Obj.write('2000A',single(23.5));
```

For the third example, write a 2-D array of integers in row-major format (standard C programming format) at address 131072 (decimal).

```
mlarr = int32([1:10;101:110]);
IDE_Obj.write(131072,mlarr');
```

See Also

hex2dec | read

xmakefilesetup

Purpose

Configure your coder product to generate makefiles

Syntax

```
xmakefilesetup
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

You can configure your coder product to generate and build your software using makefiles. This process can use the software build toolchains, such as compilers and linkers, associated with the preceding list of IDEs. However, the makefile build process does not use the graphical user interface of the IDE directly.

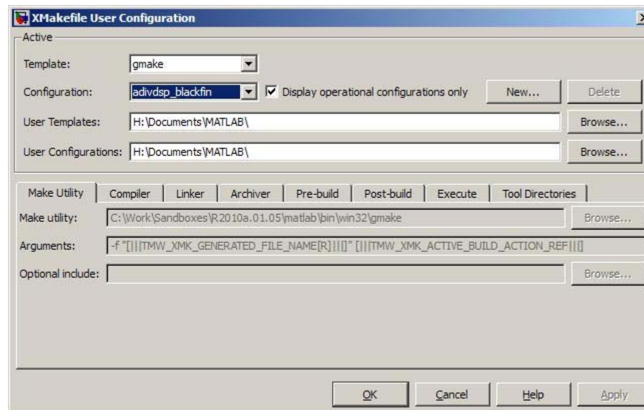
Enter `xmakefilesetup` at the MATLAB command line to configure how to generate makefiles.

Use this function:

- Before you build your software using makefiles for the first time.
- If you change the software build toolchain or processor family.

For more instructions and examples, see "Makefiles for Software Build Tool Chains".

The `xmakefile` function displays the following dialog box, which prompts you for information about your make utility and software build toolchain.



See Also

“Build format” on page 6-277 | “Build action” on page 6-279

Block Reference

Asynchronous (p. 4-2)	Specify asynchronous function-call inputs or create interrupt support blocks
Custom Code (p. 4-3)	Insert custom code into generated model files and subsystem functions
Desktop Targets (desktoptargetslib) (p. 4-4)	Generate code for execution on host Windows or Linux systems
S-Function Target (p. 4-6)	Generate code for S-function

Asynchronous

Asynchronous Task Specification	Allow for parameter specifications for asynchronous tasks associated with root-level Inport blocks that output a function-call trigger
Interrupt Templates (p. 4-2)	Create blocks that provide interrupt support for real-time operating system (RTOS)

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

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

Desktop Targets (desktoptargetslib)

In this section...

"Host Communication" on page 4-4
"Target Preferences" on page 4-4
"Linux" on page 4-4
"Windows" on page 4-5

Host Communication

Byte Pack	Convert input signals to `uint8` vector
Byte Reversal	Reverse order of bytes in input word
Byte Unpack	Unpack UDP `uint8` input vector into Simulink data type values
UDP Receive	Receive UDP packet
UDP Send	Send UDP message

Target Preferences

Target Preferences	Configure model for specific IDE, tool chain, board, and processor

Linux

Linux Audio Capture	Capture ALSA audio from sound card and output data
Linux Audio Playback	Send audio data stream to ALSA audio device output
Linux Task	Spawn task function as separate Linux thread

UDP Receive	Receive UDP packet
UDP Send	Send UDP message

Windows

UDP Receive	Receive UDP packet
UDP Send	Send UDP message
Windows Task	Spawn task function as separate Windows thread

S-Function Target

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

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

Asynchronous / Interrupt Templates

Description

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

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

You can use the block for simulation and code generation.

Parameters

VME interrupt number(s)

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

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

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

VME interrupt vector offset(s)

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

Simulink task priority(s)

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

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

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

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

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

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

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

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

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

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

Timer size
The number of bits to be used to store the clock tick for a hardware timer. The ISR generated by the Async Interrupt block uses the timer size when you select **Manage own timer**. The size can be

`32bits` (the default), `16bits`, `8bits`, or `auto`. If you select `auto`, the Simulink Coder software determines the timer size based on the settings of **Application lifespan (days)** and **Timer resolution**.

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

For more information, see “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 prevent the blocks from contributing to the generated code. Alternatively, you can use the Environment Controller block, as explained in “Dual-Model Approach: Code Generation”. However, if you use the Environment Controller block, be aware that the sample times of driving blocks contribute to the sample times supported in the generated code.

Inputs and Outputs

Input

A simulated interrupt source.

Output

Control signal for a

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

Assumptions and Limitations

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

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

Performance Considerations

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

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

See Also

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

Asynchronous Task Specification

Purpose

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

Library

Asynchronous

Description

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

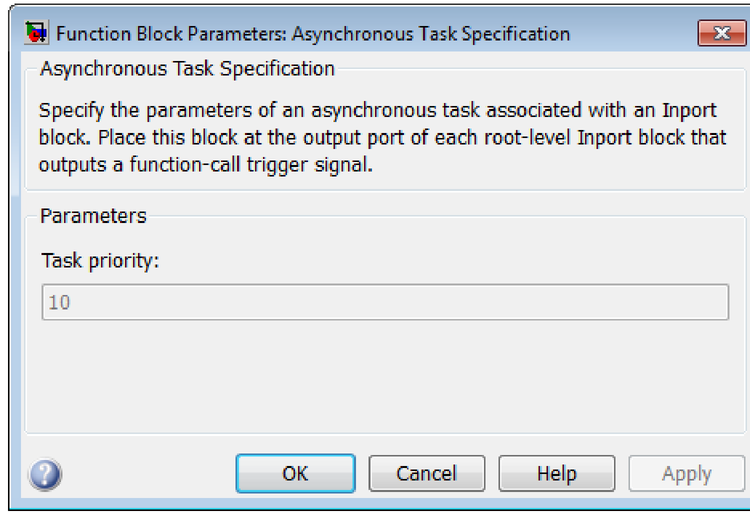
To implement this feature, place this block at the output port of each root-level Inport block that outputs a function-call trigger. On the **Signal Attributes** pane of the Inport block, select **Output function call** to specify that the Inport block accepts function-call signals. Then use the Asynchronous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

Data Type Support

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

Parameters and Dialog Box

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



Task priority

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

Settings

Default: `10`

- You can enter an integer or [].
- If you specify an integer for an Asynchronous Task Specification block that resides in a model reference, then the initiator in the top model must have the same integer value for its priority.
- If you specify [] for an Asynchronous Task Specification block that resides in a model reference, then the initiator in the top model can have any priority. For this case, the rate transition algorithm is conservative (not optimized), assuming that the priority is unknown but static.

Command-Line Information

This block has only one parameter.

Parameter: `TaskPriority`

Value: `integer`

Configuration Parameters Settings

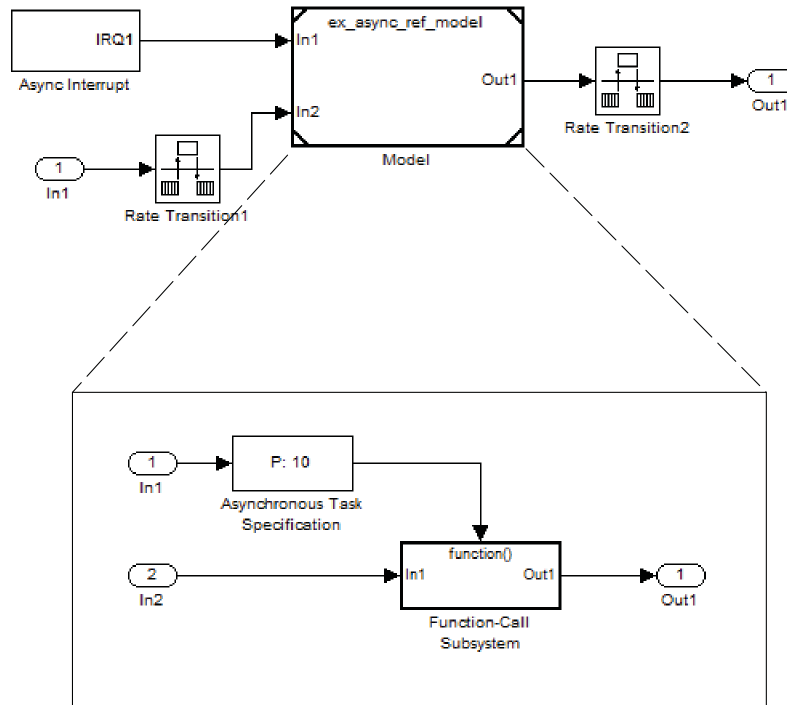
To create an asynchronous model reference containing a Function-Call and an Asynchronous Task Specification block, you must follow the procedure outlined in "Converting an Asynchronous Subsystem into a Model Reference". One of the steps requires that you make several changes to configuration parameters.

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

Examples

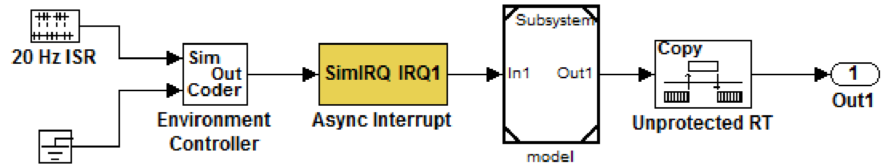
Asynchronous Function-Call Input to Model

This root-level model uses the Inport block with the Asynchronous Task Specification block to allow a function-call input signal to a model reference. The priority is set to 10.

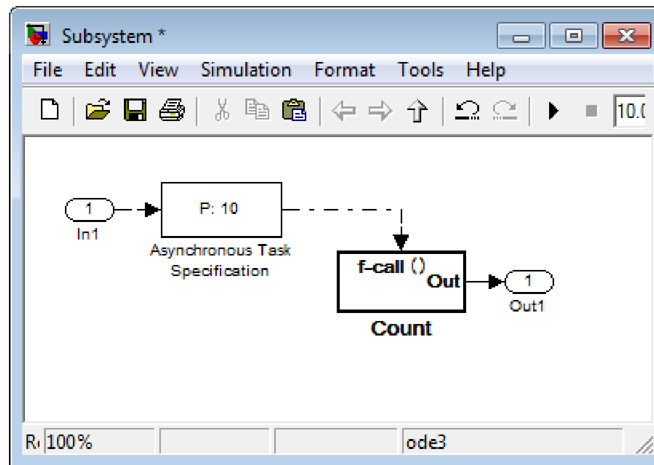


The Asynchronous Task Specification block must immediately follow the Inport block. Also, no branch can emanate from the signal connecting the Inport block to the Asynchronous Task Specification block.

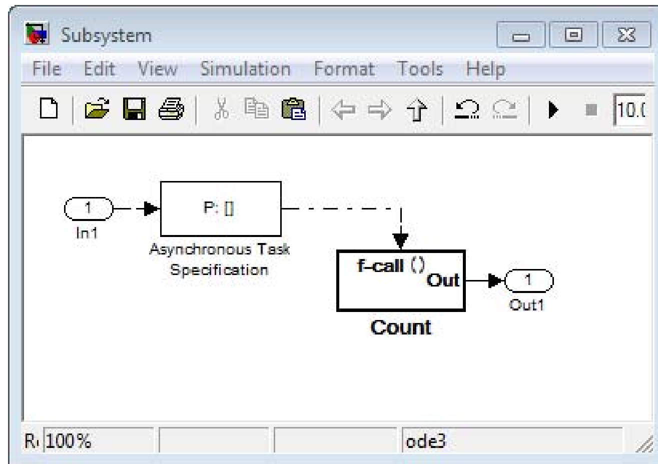
Setting Priorities



For this model, if the Asynchronous Task Specification block is set to the default value of 10, then the Async Interrupt block must also have a priority of 10.



Whereas, if the priority of the Asynchronous Task Specification block is set to the empty matrix, [], then the Async Interrupt can have any value.



Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	N/A
Dimensionalized	No
Multidimensionalized	No
Zero-Crossing Detection	No

See Also

Function-Call Subsystem block

“Handling Asynchronous Events”

“Referencing a Model”

Inport block

Byte Pack

Purpose

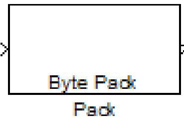
Convert input signals to `uint8` vector

Library

Embedded Coder/ Embedded Targets/ Host Communication

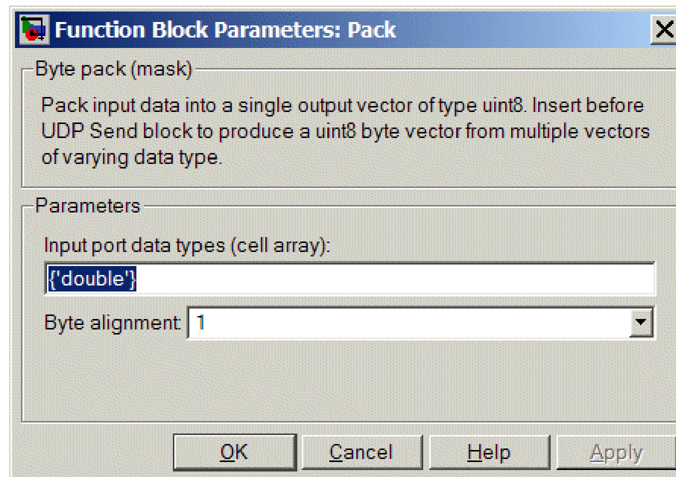
Simulink Coder/ Desktop Targets/ Host Communication

Description



Using the input port, the block converts data of one or more data types into a single `uint8` vector for output. With the options available, you specify the input data types and the alignment of the data in the output vector. Because UDP messages are in `uint8` data format, use this block before a UDP Send block to format the data for transmission using the UDP protocol.

Dialog Box



Input port data types (cell array)

Specify the data types for the different signals as part of the parameters. The block supports all Simulink data types except characters. Enter the data types as Simulink types in the cell array, such as '`double`' or '`int32`'. The order of the data type entries in the cell array must match the order in which the data arrives at the block input. This block determines the signal sizes

automatically. The block always has at least one input port and only one output port.

Byte alignment

This option specifies how to align the data types to form the `uint8` output vector. Select one of the values in bytes from the list.

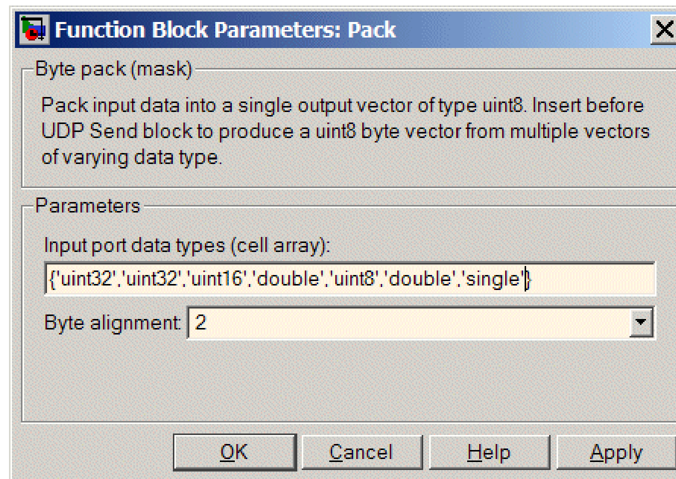
Alignment can occur on 1, 2, 4, or 8-byte boundaries depending on the value you choose. The value defaults to `1`. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithm ensures that each element in the output vector begins on a byte boundary specified by the alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

Selecting 1 for **Byte alignment** provides the tightest packing, with no holes between any data types for any combination of data types and signals.

Sometimes, you can have multiple data types of varying lengths. In such cases, specifying a 2-byte alignment can produce 1–byte gaps between uint8 or int8 values and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time. You can specify any of the data alignment options with any of the data types.

Example

Use a cell array to enter input data types in the **Input port data types** parameter. The order of the data types you enter must match the order of the data types at the block input.



In the cell array, you provide the order in which the block expects to receive data—`uint32`, `uint32`, `uint16`, `double`, `uint8`, `double`, and `single`. With this information, the block automatically provides the proper number of input ports.

Byte alignment equal to `2` specifies that each new value begins 2 bytes from the previous data boundary.

The example shows the following data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

When the signals are scalar values (no matrices or vectors in this example), the first signal value in the vector starts at 0 bytes. Then, the second signal value starts at 2 bytes, and the third at 4 bytes. Next, the fourth signal value follows at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. As the example shows, the packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

See Also

Byte Reversal, Byte Unpack

Purpose

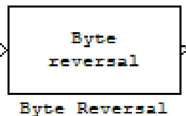
Reverse order of bytes in input word

Library

Embedded Coder/ Embedded Targets/ Host Communication

Simulink Coder/ Desktop Targets/ Host Communication

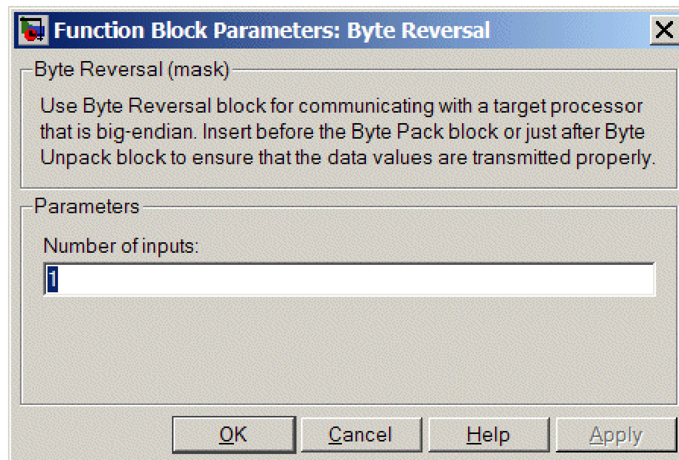
Description



Byte reversal changes the order of the bytes in data you input to the block. Use this block when your process communicates between targets that use different endianness, such as between Intel® processors that are little endian and others that are big endian. Texas Instruments™ processors are little-endian by default.

To exchange data with a processor that has different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

Dialog Box



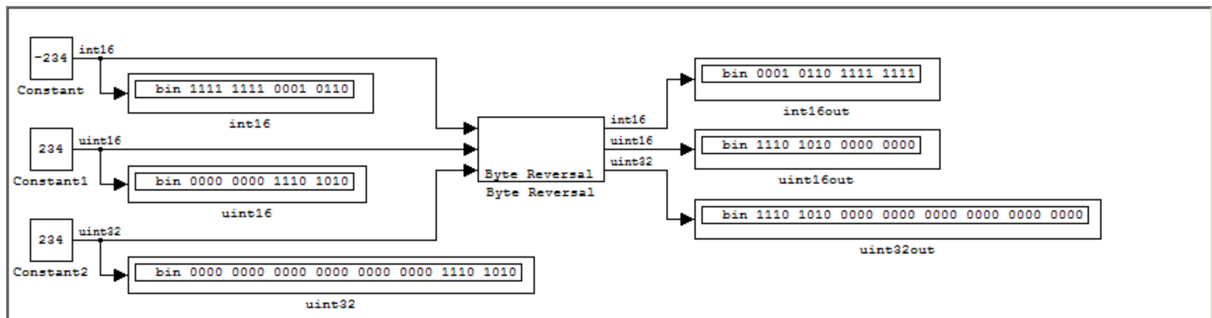
Number of inputs

Specify the number of input ports for the block. The number of input ports adjusts automatically to match value so the number of outputs equals the number of inputs.

When you use more than one input port, each input port maps to the matching output port. Data entering input port 1 leaves through output port 1, and so on.

Reversing the bytes does not change the data type. Input and output retain matching data type.

The following model shows byte reversal in use. In this figure, the input and output ports match for each path.



See Also

Byte Pack, Byte Unpack

Purpose

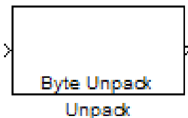
Unpack UDP `uint8` input vector into Simulink data type values

Library

Embedded Coder/ Embedded Targets/ Host Communication

Simulink Coder/ Desktop Targets/ Host Communication

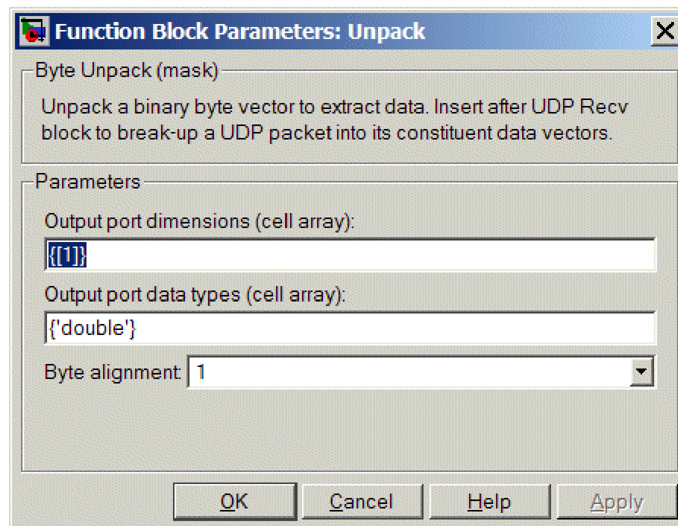
Description



Byte Unpack is the inverse of the Byte Pack block. It takes a UDP message from a UDP receive block as a `uint8` vector, and outputs Simulink data types in various sizes depending on the input vector.

The block supports all Simulink data types.

Dialog Box



Output port dimensions (cell array)

Containing a cell array, each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding Byte Pack block in the model.

Entering one value means that the block applies that dimension to all data types.

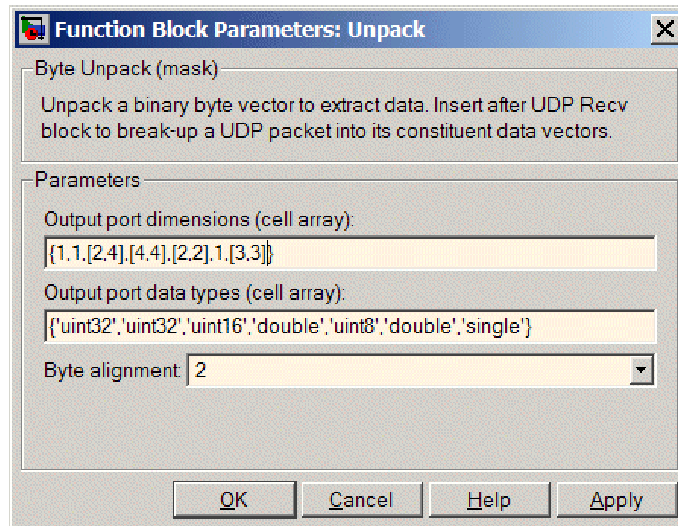
Output port data types (cell array)
Specify the data types for the different input signals to the Pack block. The block supports all Simulink data types—`single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`, and `Boolean`. The entry here is the same as the Input port data types parameter in the Byte Pack block in the model. You can enter one data type and the block applies that type to all output ports.

Byte Alignment
This option specifies how to align the data types to form the input `uint8` vector. Match this setting with the corresponding Byte Pack block alignment value of `1`, `2`, `4`, or `8` bytes.

Example

This figure shows the Byte Unpack block that corresponds to the example in the Byte Pack example. The **Output port data types (cell array)** entry shown is the same as the **Input port data types (cell array)** entry in the Byte Pack block

```
{'uint32','uint32','uint16','double','uint8','double','single'}.
```



In addition, the **Byte alignment** setting matches as well. **Output port dimensions (cell array)** now includes scalar values and matrices to demonstrate entering nonscalar values. The example for the Byte Pack block assumed only scalar inputs.

See Also

Byte Pack, Byte Reversal

Generated S-Function

Purpose

Represent model or subsystem as generated S-function code

Library

S-Function Target

Description

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

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

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

Requirements

- The S-Function block must perform identically to the model or subsystem from which it was generated.
- Before creating the block, 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 Simulink Coder documentation.
- You must set the solver parameters of the Generated S-Function block to be the same as those of the original model or subsystem. The generated S-function code will operate identically to the original subsystem (see Choice of Solver Type in the Simulink Coder documentation for an exception to this rule).

Parameters

Generated S-function name (model_sf)

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

Show module list

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

See Also

"Creating an S-Function Block from a Subsystem" in the Simulink Coder documentation

Linux Audio Capture

Purpose

Capture ALSA audio from sound card and output data

Library

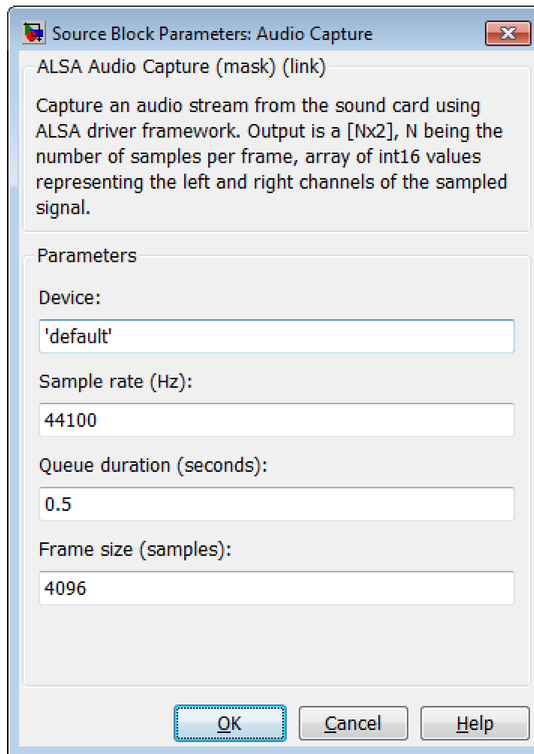
Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

Simulink Coder/ Desktop Targets/ Operating Systems/ Linux

Description

This block uses the ALSA driver framework to capture an audio stream from a sound card. It outputs the left and right channels of the signal as an [Nx2] frame of int16 values. N is the number of samples per frame.

Dialog



Device

Use the default ALSA device, or enter the name of a specific audio output device.

Entering `'default'` selects the ALSA device specified by an ALSA configuration file on your target Linux® system.

One of the following ALSA configuration files defines the default device:

- `/etc/asound.conf`, which defines system-wide options for all users
- `~/.asoundrc`, which overrides `/etc/asound.conf` for the current user

The entry that specifies the default device looks similar to this example:

```
pcm.!default {
    type hw
    card 0
    device 2
}
```

To enter the name of an alternate audio input device, review the `/proc/asound/cards` file on your target Linux system. For example, if `/proc/asound/cards` contained the following entries, you could set the value of **Device** to `'AudioPCI'` :

```
$ cat /proc/asound/cards

0 [Dummy      ]: Dummy - Dummy
                 Dummy 1

1 [VirMIDI    ]: VirMIDI - VirMIDI
                 Virtual MIDI Card 1
```

```
2 [AudioPCI   ]: ENS1371 - Ensoniq AudioPCI
                 Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

The default value for **Device** is `'default'`.

Sample rate (Hz)

Enter a value that matches the sample rate of the ALSA audio output.

By default, the sample rate of the ALSA output equals the output of the audio capture device. In this case, enter the sample rate of the audio capture device.

The `/etc/asound.conf` and `~/.asoundrc` files can configure ALSA to downsample the signal from the audio capture device. In this case, enter the downsample rate specified by the configuration files. For example, if one of the configuration files contained the following entry, you would set the value of **Sample rate (Hz)** to `16000` :

```
pcm_slave.sl3 {
    pcm ens1371
    format S16_LE
    channels 1
    rate 16000
}
pcm.complex_convert {
    type plug
    slave sl3
}
```

The default value for **Sample rate (Hz)** is `44100` Hz (44.1 kHz). The maximum rate equals the sampling rate of the audio capture device.

Queue duration (seconds)

Set the duration of the queue in seconds. This queue provides a software-based frame buffer between the ALSA output and the

Linux Audio Capture block. The queue prevents dropped data caused by temporary mismatches in the rate of data arriving and leaving the queue. Higher values can handle more significant mismatches, but such values also increase signal latency and memory usage.

The default value for **Queue duration (seconds)** is `0.5` seconds.

Frame size (samples)

Set the number of samples per frame in the output this block sends to your model. The default value for this parameter is `4096` samples.

References

http://www.alsa-project.org

See Also

http://www.alsa-project.org

Linux Audio Playback

Linux Task

Linux Audio Playback

Purpose

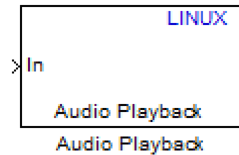
Send audio data stream to ALSA audio device output

Library

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux (linuxlib)

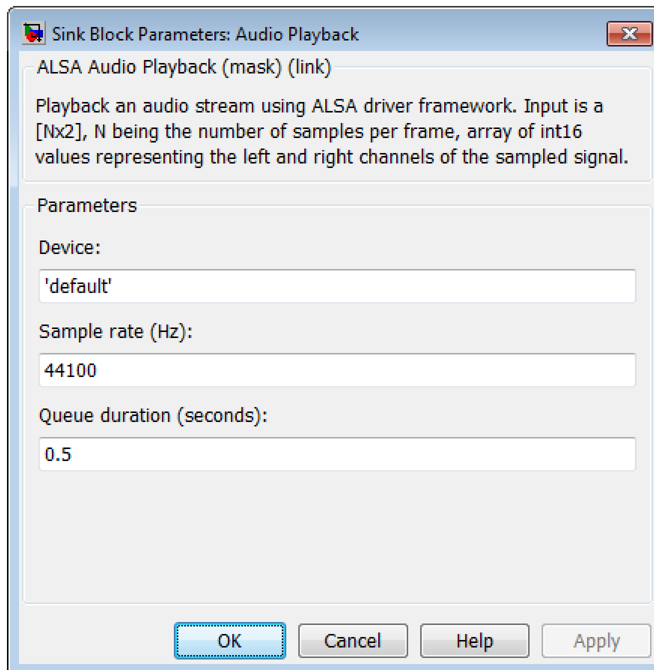
Simulink Coder/ Desktop Targets/ Operating Systems/ Linux

Description



This block takes a stream of audio data and sends it to the output jack of an ALSA audio device. The block input, **In**, takes the left and right channels of data as an [Nx2] frame of int16 values. N is the number of samples per frame.

Dialog



Device

Use the default ALSA device, or enter the name of a specific audio device.

Entering `'default'` selects the ALSA device specified by an ALSA configuration file on your target Linux system.

One of the following ALSA configuration files defines the default device:

- `/etc/asound.conf`, which defines system-wide options for all users
- `~/.asoundrc`, which overrides `/etc/asound.conf` for the current user

The entry that specifies the default device looks like this hypothetical example:

```
pcm.!default {
    type hw
    card 0
    device 2
}
```

To enter the name of an alternate audio device, consult the `/proc/asound/cards` file on your target Linux system. For example, if `/proc/asound/cards` contained the following hypothetical entries, you could set the value of **Device** to `'AudioPCI'`:

```
$ cat /proc/asound/cards

0 [Dummy      ]: Dummy - Dummy
                 Dummy 1

1 [VirMIDI    ]: VirMIDI - VirMIDI
                 Virtual MIDI Card 1
```

```
2 [AudioPCI    ]: ENS1371 - Ensoniq AudioPCI
                  Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

The default value for **Device** is `'default'`.

Sample rate (Hz)

Enter a value that matches the sample rate of the ALSA audio output.

By default, the sample rate of the ALSA output is the same as the output of the audio capture device. In this case, enter the sample rate of the audio capture device.

The `/etc/asound.conf` and `~/.asoundrc` files can configure ALSA to downsample the signal from the audio capture device. In this case, enter the downsample rate specified by the configuration files. For example, if one of the configuration files contained the following hypothetical entry, you would set the value of **Sample rate (Hz)** to `16000` :

```
pcm_slave.sl3 {
    pcm ens1371
    format S16_LE
    channels 1
    rate 16000
}
pcm.complex_convert {
    type plug
    slave sl3
}
```

The default value for **Sample rate (Hz)** is `44100` Hz (44.1 kHz). The maximum rate is the sampling rate of the audio capture device.

Queue duration (seconds)

Set the duration of the queue in seconds. This queue provides a software-based frame buffer between the ALSA audio device and this block. The queue prevents dropped data caused by temporary mismatches in the rate of data arriving and leaving the queue. Higher values can handle more significant mismatches, but increase signal latency and memory usage.

The default value for **Queue duration (seconds)** is `0.5` seconds.

See Also

http://www.alsa-project.org

Linux Audio Capture

Linux Task

Linux Task

Purpose

Spawn task function as separate Linux thread

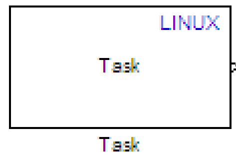
Library

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

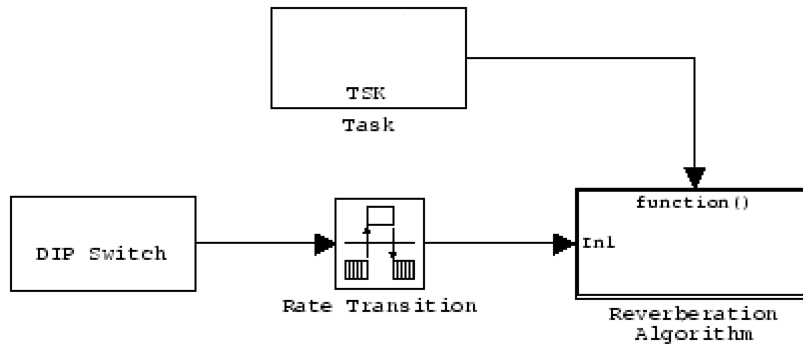
Simulink Coder/ Desktop Targets/ Operating Systems/ Linux

Description

This documentation will be updated.

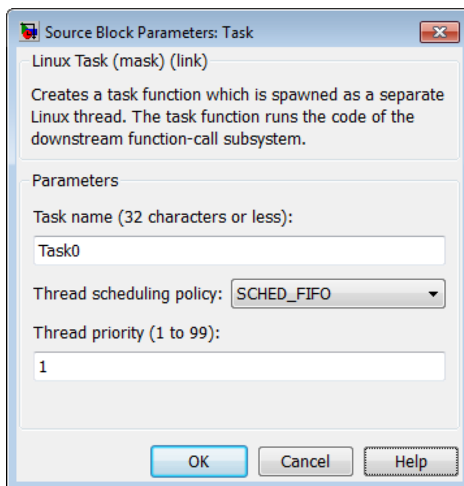


Use this block to create a task function that spawns as a separate Linux thread. The task function runs the code of the downstream function-call subsystem. For example:



Dialog

This documentation will be updated.



Task name

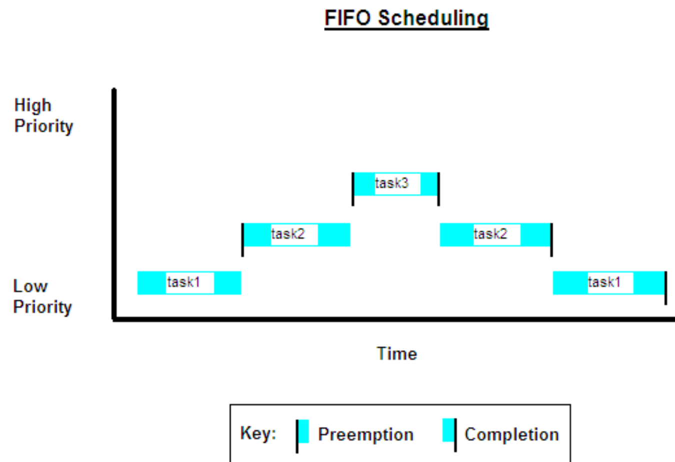
Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

Thread scheduling policy

Select the scheduling policy that applies to this thread. You can choose from the following options:

- `SCHED_FIFO` enables a First In, First Out scheduling algorithm that executes real-time processes without time slicing. With FIFO scheduling, a higher-priority process preempts a lower-priority process. The lower-priority process remains at the top of the list for its priority and resumes execution when the scheduler blocks all higher-priority processes.

  For example, in the following image, task2 preempts task1. Then task3 preempts task2. When task3 completes, task2 resumes. When task2 completes, task1 resumes.



Selecting SCHED_FIFO, displays the **Thread priority** parameter, which you can set to a value from 1 to 99.

- SCHED_OTHER enables the default Linux time-sharing scheduling algorithm. You can use this scheduling for all processes except those requiring special static priority real-time mechanisms. With this algorithm, the scheduler chooses processes based on their dynamic priority within the static priority 0 list. Each time the process is ready to run and the scheduler denies it, the operating system increases that process's dynamic priority. Such prioritization ensures the scheduler serves the SCHED_OTHER processes in the correct order.

  Selecting SCHED_OTHER, hides the **Thread priority** parameter, and sets the thread priority to 0.

Thread priority (1 to 99)

When you set **Thread scheduling policy** to SCHED_FIFO, you can set the priority of the thread from 1 to 99 (low-to-high).

Higher-priority tasks can preempt lower-priority tasks.

See Also

Linux Audio Capture

Linux Audio Playback

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

Parameters

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

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

Example

See "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
"Custom Code Block Code Insertion" in the Simulink Coder documentation

Purpose Specify custom source code

Library Custom Code

Description The Model Source block adds user-specified custom code to the `model.c` or `model.cpp` file that 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

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

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

Example See "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
"Custom Code Block Code Insertion" in the Simulink Coder documentation

Protected RT

Purpose

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

Library

VxWorks (`vxlib1`)

Description

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

Purpose

Specify custom system derivative code

Library

Custom Code

Description

The System Derivatives block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemDerivatives` function that 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Derivatives Function Declaration Code
Code to be added to the declaration section of the generated `SystemDerivatives` function.

System Derivatives Function Execution Code
Code to be added to the execution section of the generated `SystemDerivatives` function.

System Derivatives Function Exit Code
Code to be added to the exit section of the generated `SystemDerivatives` function.

Example

See “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
“Custom Code Block Code Insertion” in the Simulink Coder 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

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

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

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

Example See “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
“Custom Code Block Code Insertion” in the Simulink Coder documentation

Purpose

Specify custom system enable code

Library

Custom Code

Description

The System Enable block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemEnable` function that 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Enable Function Declaration Code
Code to be added to the declaration section of the generated `SystemEnable` function.

System Enable Function Execution Code
Code to be added to the execution section of the generated `SystemEnable` function.

System Enable Function Exit Code
Code to be added to the exit section of the generated `SystemEnable` function.

Example

See “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
“Custom Code Block Code Insertion” in the Simulink Coder 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

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

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

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

Example

See “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
“Custom Code Block Code Insertion” in the Simulink Coder documentation

Purpose

Specify custom system outputs code

Library

Custom Code

Description

The System Outputs block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemOutputs` function that 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Outputs Function Declaration Code
Code to be added to the declaration section of the generated `SystemOutputs` function.

System Outputs Function Execution Code
Code to be added to the execution section of the generated `SystemOutputs` function.

System Outputs Function Exit Code
Code to be added to the exit section of the generated `SystemOutputs` function.

Example

See "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
"Custom Code Block Code Insertion" in the Simulink Coder 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

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

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

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

Example

See “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
“Custom Code Block Code Insertion” in the Simulink Coder documentation

Purpose

Specify custom system termination code

Library

Custom Code

Description

The System Terminate block adds user-specified custom code to the declaration, execution, and exit code sections of the `SystemTerminate` function that 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

System Terminate Function Declaration Code
Code to be added to the declaration section of the generated `SystemTerminate` function.

System Terminate Function Execution Code
Code to be added to the execution section of the generated `SystemTerminate` function.

System Terminate Function Exit Code
Code to be added to the exit section of the generated `SystemTerminate` function.

Example

See “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
“Custom Code Block Code Insertion” in the Simulink Coder 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 Simulink Coder build process ignores the block for simulation target builds, but includes any specified custom code in the build process for other targets.

Parameters

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

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

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

Example

See “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
“Custom Code Block Code Insertion” in the Simulink Coder documentation

Purpose

Configure model for specific IDE, tool chain, board, and processor

Library

Simulink Coder/ Desktop Targets

Embedded Coder/ Embedded Targets

Description

Use the Target Preferences block to configure a model to for a specific IDE/tool chain, board, and processor. Your MathWorks® software depends on this information to properly simulate the model and generate code for your environment.

The appearance and contents of the Target Preferences block varies widely, depending on the options you have selected. The following sections describe all of the user interface elements in the Target Preferences block, even though the Target Preferences block cannot simultaneously display all of the user interface elements.

For more information, see the Target Preferences topic in the User's Guide.

Note The following actions update the appropriate model Configuration Parameters with new values:

- Adding a Target Preferences block to your model and clicking Yes in the **Initialize Configuration Parameters** dialog box.
- Opening the Target Preferences block in your model and selecting a new **IDE/Tool Chain**.
- Opening the Target Preferences block in your model and applying changes to the **Board** and **Processor** parameters.

Note If you are using a Windows host, use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Note The figures in this documentation include references to various third-party vendors and products. These images aid with recognition of specific user interface elements. Do not infer a preference or endorsement for any vendor or product over another.

Dialog Boxes

This reference page section contains the following subsections:

- “Board Pane” on page 5-49
- “Add Processor Dialog Box” on page 5-53
- “Linux Pane” on page 5-54
- “Windows Pane” on page 5-55

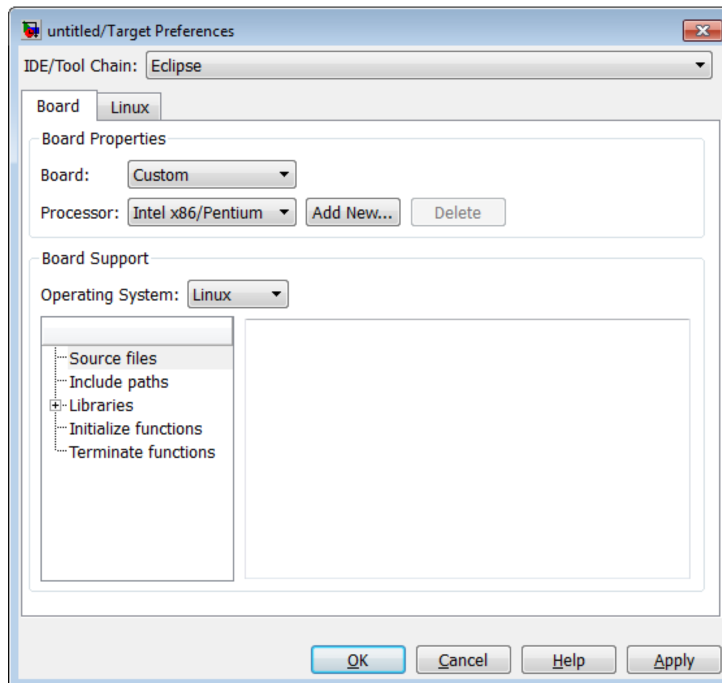
Use the **IDE/Tool Chain** parameter to select the Integrated Development Environment (IDE) or software build tool chain with which you are working. Selecting any option automatically applies that selection to the Target Preferences block and updates the panes and options the block displays.

Target Preferences block dialog box provides tabbed access to the following panes:

- Board Pane — Select the target board, processor, clock speed, and, in some cases, RTOS. In addition, **Add new** on this pane opens the **Add Processor** dialog box.
- Memory Pane — Set the memory allocation and layout on the processor (memory mapping).

- Section Pane — Determine the arrangement and location of the sections on the processor and compiler information.
- Linux Pane — For the Eclipse IDE: Specify the scheduling mode and base rate task priority of the software to run on a Linux target.
- Windows Pane — For the Eclipse IDE: Specify the scheduling mode of the software to run on a Windows target.
- VxWorks Pane — For the Wind River Diab/GCC (makefile generation only): Specify the scheduling mode of the software to run on a VxWorks target.

Board Pane



The following options appear on the **Board** pane, which has separate panels for **Board Properties**, **Board Support**, and **IDE Support** labels.

Board

Select your target board from the list of options. Selecting a specific board sets the appropriate value for the **Processor** parameter. If you select a custom board, also set the **Processor** parameter to an appropriate value.

Processor

The Board and Processor settings apply default values to many of the remaining Target Preferences parameters, such as those under the **Memory** and **Section** tabs.

If the coder product supports an operating system for the processor, it enables the **Operating system** option.

If you are using the Eclipse IDE and set **Processor** to `Generic/Custom`, open the model Configuration Parameters and use the **Hardware Implementation** pane to define the custom hardware. With this approach, hardware support depends on the Simulink Coder product, not on the coder product. For more information, see “Hardware Implementation Pane”.

Note Selecting or reselecting a processor resets the solver and some processor-specific parameters to their default values.

Add New

Clicking **Add new** opens a new dialog box where you specify configuration information for a processor that is not on the Processor list.

For details about the New Processor dialog box, refer to “Add Processor Dialog Box” on page 5-53.

Delete

Delete a processor that you added to the **Processor** list. You cannot delete any of the standard processors.

CPU Clock

Enter the actual clock rate the board uses. This action does not change the rate on the board. Rather, the code generation process requires this information to produce code that runs correctly on the hardware. Setting this value incorrectly causes timing and profiling errors when you run the code on the hardware.

The timer uses the value of **CPU clock** to calculate the time for each interrupt. For example, a model with a sine wave generator block running at 1 kHz uses timer interrupts to generate sine wave samples at the proper rate. For example, using 100 MHz, the timer calculates the sine generator interrupt period as follows:

- Sine block rate = 1 kHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires:

100,000,000/1000 = 1 Sine block interrupt per 100,000 clock ticks

Board Support

Select the following parameters and edit their values in the text box on the right:

- `Source files` — Enter the full paths to source code files.
- `Include paths` — Add paths to include files.
- `Libraries` — Identify specific libraries for the processor. Required libraries appear on the list by default. To add more libraries, entering the full path to the library with the library file in the text area.

- `Initialize functions` — If your project requires an initialize function, enter it in this field. By default, this parameter is empty.
- `Terminate functions` — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

Note Invalid or incorrect entries in these fields can cause errors during code generation. When you enter a file path, library, or function, the block does not verify that the path or function exists or is valid.

When entering a path to a file, library, or other custom code, use the following string in the path to refer to the IDE installation folder.

```
$(Install_dir)
```

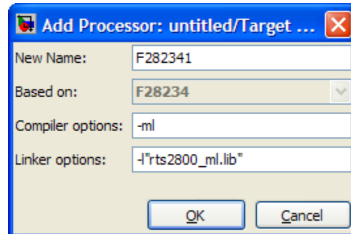
Enter new paths or files (custom code items) one entry per line. Include the full path to the file for libraries and source code. **Support** options do not support functions that use return arguments or values. These parameters accept only functions of type `void fname void` as valid as entries.

You can also set up environment variables to use as folder path tokens. For example, if you set up an environment called `USER_VAR`, you can use it as a token when you define a path in your Target Preferences block. For example: `$(USER_VAR)\myinstal\foo.c`.

Operating System

Select an operating system or RTOS for your target. If your target platform supports an operating system, the software enables the **Operating system** parameter. Otherwise, the software disables this option.

Add Processor Dialog Box



To add a new processor to the drop down list for the **Processors** option, click the **Add new** button on the **Board** pane. The software opens the **Add Processor** dialog box.

Note You can use this feature to create duplicates of existing processors with minor changes to the compiler and linker options. Avoid using this feature to create profiles for processors that are not already supported.

New Name
: Provide a name to identify your new processor. Use any valid C string. The name you enter in this field appears on the list of processors after you add the new processor.

  If you do not provide an entry for each parameter, the coder product returns an error message without creating a processor entry.

Based On
: When you add a processor, the dialog box uses the settings from the currently selected processor as the basis for the new one. This parameter displays the currently selected processor.

Compiler options
: Identifies the processor family of the new processor to the compiler. Successful compilation requires this switch. The string depends on the processor family or class.

Linker options

You can use this parameter to specify linker command options. The IDE uses these options to modify how it links project files when you build a project. To get information about specific linker options you can enter here, consult the documentation for your IDE.

Linux Pane

The Linux tab appears when you set **IDE/Tool Chain** to `Eclipse` and set **Operating System** on the Board tab to `Linux`.

The Linux tab displays two options:

Scheduling Mode

When you select `free-running`, the model generates multi-threaded free-running code. Each rate in the model maps to a separate thread in the generated code. Multi-threaded code can potentially run faster than single threaded code.

When you select `real-time`, the model generates multi-threaded real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example, a 1-second rate runs at exactly 1-second intervals. The timing is provided by using a Linux real-time clock.

Base rate task priority

The base rate in the model maps to a thread and runs as fast as possible. You can use the value of the base rate priority to set a static priority for the base rate task. By default, this rate is 40.

Allow tasks to execute concurrently

Note This parameter will be removed in a future release.

Enable multicore deployment. Selecting this option enables generated multi-threading code to run concurrently on multicore processors. By default, this option is disabled.

This parameter has been superseded. Configuring the model as described in the following procedures hides the **Allow tasks to execute concurrently** parameter from view.

To run target applications on multicore processors, follow the procedures in "Running Target Applications on Multicore Processors", and "Configuring Models for Targets with Multicore Processors".

Windows Pane

The Windows tab appears when you set **IDE/Tool Chain** to `Eclipse` and set **Operating System** on the Board tab to `Windows`.

The Windows tab displays one option:

Scheduling Mode

When you select `free-running`, the model generates multi-threaded free-running code. Each rate in the model maps to a separate thread in the generated code. Multi-threaded code can potentially run faster than single threaded code.

When you select `real-time`, the model generates multi-threaded real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example, a 1-second rate runs at exactly 1-second intervals. The timing is provided by using a Windows real-time clock.

Allow tasks to execute concurrently

Note This parameter will be removed in a future release.

Enable multicore deployment. Selecting this option enables generated multi-threading code to run concurrently on multicore processors. By default, this option is disabled.

This parameter has been superseded. Configuring the model as described in the following procedures hides the **Allow tasks to execute concurrently** parameter from view.

To run target applications on multicore processors, follow the procedures in "Running Target Applications on Multicore Processors", and "Configuring Models for Targets with Multicore Processors".

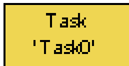
Purpose

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

Library

Asynchronous / Interrupt Templates

Description



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

The Task Sync block performs the following functions:

- Uses the VxWorks system call `taskSpawn` to spawn an independent task. When the task is activated, it calls the downstream function-call subsystem code or Stateflow chart. The block calls `taskDelete` to delete the task during model termination.
- Creates a semaphore to synchronize the connected subsystem with execution of the block.
- Wraps the spawned task in an infinite `for` loop. In the loop, the spawned task listens for the semaphore, using `semTake`. The first call to `semTake` specifies NO_WAIT. This allows the task to determine whether a second `semGive` has occurred prior to the completion of the function-call subsystem or chart. This would indicate that the interrupt rate is too fast or the task priority is too low.
- Generates synchronization code (for example, `semGive()`). This code allows the spawned task to run. The task in turn calls the connected function-call subsystem code. The synchronization code can run at interrupt level. This is accomplished through the connection between the Async Interrupt and Task Sync blocks, which triggers execution of the Task Sync block within an ISR.
- Supplies absolute time if blocks in the downstream algorithmic code require it. The time is supplied either by the timer maintained by

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

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

Parameters

Task name (10 characters or less)
The first argument passed to the VxWorks `taskSpawn` system call. VxWorks uses this name as the task function name. This name also serves as a debugging aid; routines use the task name to identify the task from which they are called.

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

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

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

determine the size by examining the generated code for the task (and 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 Simulink Coder documentation). The timer value is read at the time the asynchronous interrupt is serviced, and data transfers to blocks called by the Task Sync block and execute within the task associated with the Async Interrupt block. Therefore, data transfers are synchronized with the caller.

Timer resolution (seconds)

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

Timer size

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

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

For more information, see "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 Simulink Coder documentation

Purpose

Receive UDP packet

Library

Embedded Coder/ Embedded Targets/ Host Communication

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

Embedded Coder/ Embedded Targets/ Operating Systems/ VxWorks

Simulink Coder/ Desktop Targets/ Host Communication

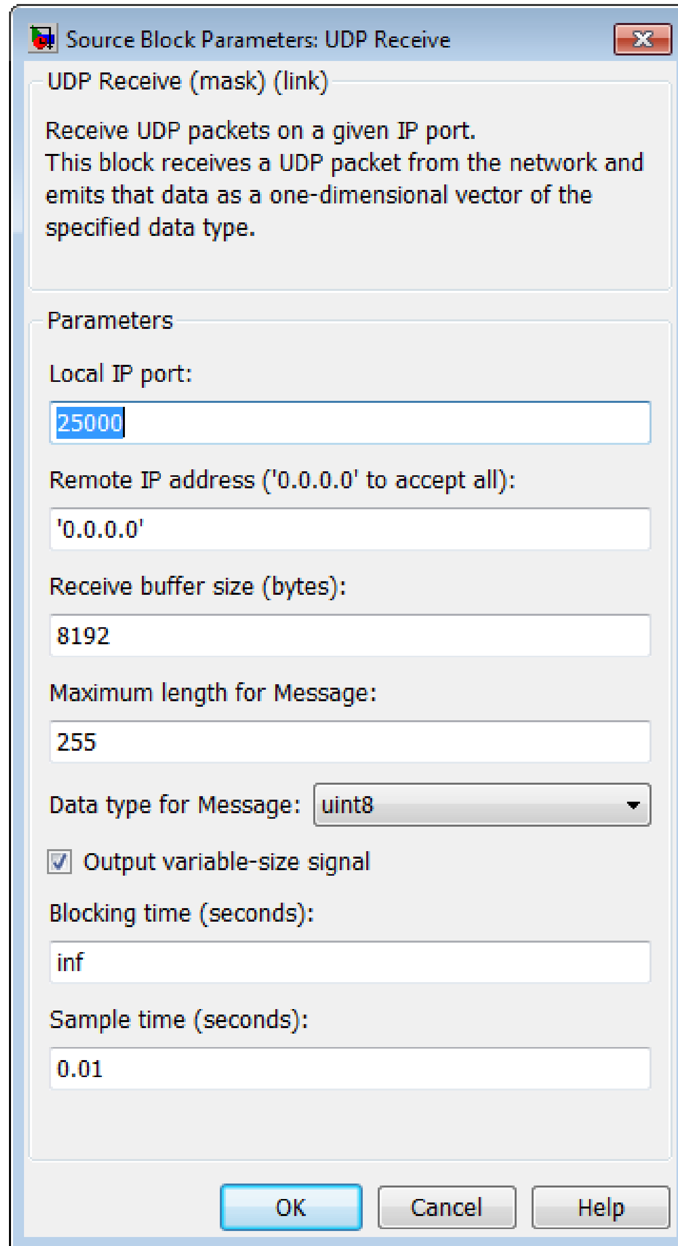
Windows (windowslib)

Note If your target system uses Linux or Windows, get the UDP block from the appropriate library, `linuxlib` or `windowslib`.

Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block output, emits the contents of a single UDP packet as a data vector.

Dialog



Local IP port

Specify the IP port number upon to receive UDP packets. This value defaults to 25000. The value can range 1–65535.

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Remote IP address ('0.0.0.0' to accept all)
Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from any other address. To accept packets from any IP address, enter `'0.0.0.0'`. This value defaults to `'0.0.0.0'`.

Receive buffer size (bytes)
Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to `8192`.

Maximum length for Message
Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of any UDP packet. The system truncates data that exceeds this length. This value defaults to `255`.

If you disable **Output variable-size signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

Data type for Message
Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Output variable-size signal
If your model supports signals of varying length, enable the **Output variable-size signal** parameter. This checkbox defaults to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Data type for Length
Set the data type of the Length output. This option defaults to `double`.

Blocking time (seconds)
For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note This parameter appears only in the UDP Receive block.

Sample time (seconds)
Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a

large value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

Output port width

Specify the width of packets the block accepts. When you design the transmit end of the UDP communication channel, you decide the packet width. Set this option to a value as large or larger than any packet you expect to receive.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

UDP receive buffer size (bytes)

Specify the size of the buffer to which the system stores UDP packets. The default size is 8192 bytes. Make the buffer large enough to store UDP packets that come in while your process reads a packet from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Send

UDP Send

Purpose

Send UDP message

Library

Embedded Coder/ Embedded Targets/ Host Communication

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

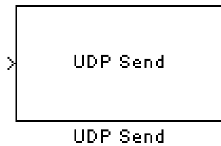
Embedded Coder/ Embedded Targets/ Operating Systems/ VxWorks

Simulink Coder/ Desktop Targets/ Host Communication

Windows (windowslib)

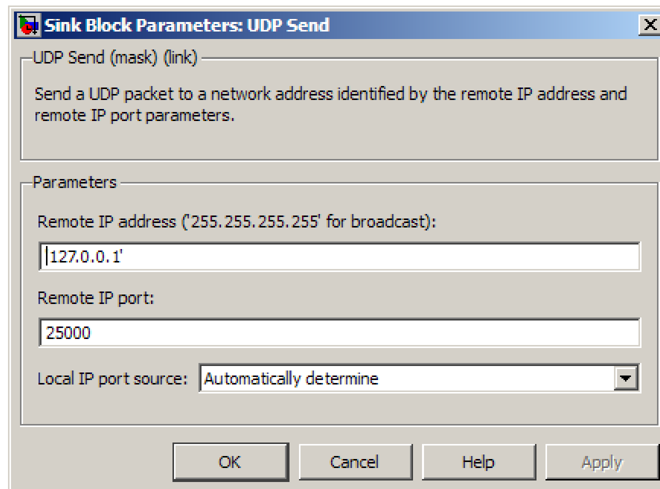
Note If your target system uses Linux or Windows, get the UDP block from the appropriate library, `linuxlib` or `windowslib`.

Description



The UDP Send block transmits an input vector as a UDP message over an IP network port.

Dialog Box



IP address ('255.255.255.255' for broadcast)

Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, `'255.255.255.255'`.

Remote IP port

Specify the port to which the block sends the message. The value defaults to 25000, but the values range from 1–65535.

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Local IP port source

To let the system automatically assign the port number, select `Assign automatically`. To specify the IP port number using the **Local IP port** parameter, select `Specify`.

Local IP port

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

Sample time

Sample time tells the block how long to wait before polling for new messages.

Note This parameter only appears in a deprecated version of the UDP Send block. Replace the deprecated UDP Send block with a current UDP Send block.

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Receive

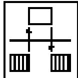
Purpose

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

Library

VxWorks (`vxlib1`)

Description



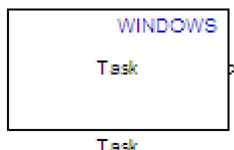
The Unprotected RT block is a Rate Transition block that is preconfigured to conduct deterministic data transfers. For more information, see Rate Transition in the SimulinkVxWorks Reference.

Windows Task

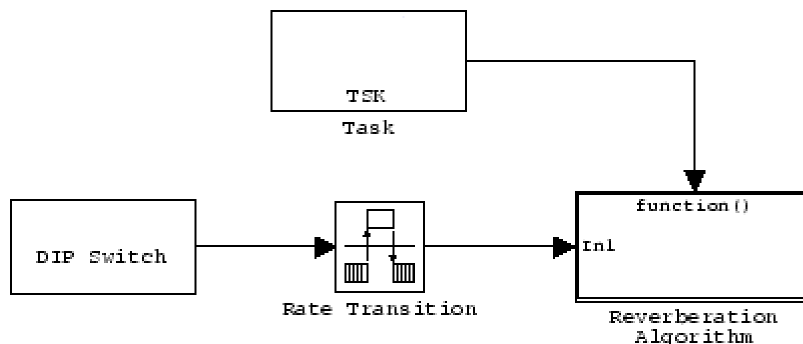
Purpose Spawn task function as separate Windows thread

Library Windows (windowslib)

Description



This block spawns a task function as a separate Windows thread. The task function runs the code of the downstream function-call subsystem. For example:



Thread priority in Windows operating systems ranges from 0 to 31 (low-to-high priority). The following two criteria determine the priority of a given thread:

- Priority class
- Priority level within the priority class

The priority classes in Windows are as follows:

- IDLE_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS

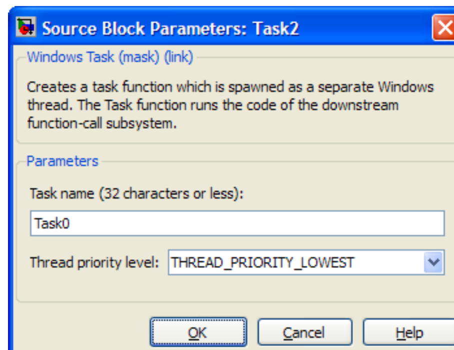
- NORMAL_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- REALTIME_PRIORITY_CLASS

This Windows Task block always uses a process priority of NORMAL_PRIORITY_CLASS.

In this Windows Task block, you can use the **Thread priority level** parameter specify the following the priority levels within in the NORMAL_PRIORITY_CLASS:

- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_ABOVE_NORMAL
- THREAD_PRIORITY_HIGHEST

Dialog



Task name

Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

Thread priority level

Set the priority for the thread. Higher-priority tasks can preempt lower-priority tasks.

Select one of the following five priority classes:

- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_ABOVE_NORMAL
- THREAD_PRIORITY_HIGHEST

See Also

6

Configuration Parameters for Simulink Models

- "Code Generation Pane: General" on page 6-2
- "Code Generation Pane: Report" on page 6-38
- "Code Generation Pane: Comments" on page 6-61
- "Code Generation Pane: Symbols" on page 6-85
- "Code Generation Pane: Custom Code" on page 6-123
- "Code Generation Pane: Debug" on page 6-140
- "Code Generation Pane: Interface" on page 6-150
- "Code Generation Pane: RSim Target" on page 6-233
- "Code Generation Pane: S-Function Target" on page 6-239
- "Code Generation Pane: Tornado Target" on page 6-245
- "Code Generation Pane: IDE Link" on page 6-274
- "Parameter Reference" on page 6-308

Code Generation Pane: General

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

Target selection
System target file: grt.tlc Browse...
Language: C
Build process
Compiler optimization level: Optimizations off (faster builds)
TLC options:
Makefile configuration
Generate makefile
Make command: make_rtw
Template makefile: grt_default_tmf
Code Generation Advisor
Select objective: Unspecified
Check model before generating code: Off Check model ...
Generate code only Build

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

Target selection

System target file: ert.tlc Browse...

Language: C

Description: Embedded Coder

Build process

Compiler optimization level: Optimizations off (faster builds)

TLC options:

Makefile configuration

☑ Generate makefile

Make command: make_rtw

Template makefile: ert_default_tmf

Data specification override

☐ Ignore custom storage classes ☐ Ignore test point signals

Code Generation Advisor

Prioritized objectives: Unspecified Set objectives ...

Check model before generating code: Off Check model ...

☐ Generate code only Build

In this section...

"Code Generation: 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
"Set Objectives — Code Generation Advisor Dialog Box" on page 6-29
"Check model" on page 6-32
"Check model before generating code" on page 6-33
"Generate code only" on page 6-35
"Build/Generate code" on page 6-37

Code Generation: General Tab Overview

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

See Also

"Code Generation Pane: General" on page 6-2

System target file

Specify the system target file.

Settings

Default: grt.tlc

You can specify the system target file in these ways:

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

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 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 Embedded Coder license)

See Also

"Available Targets"

Language

Specify C or C++ code generation.

Settings

Default: `C`

`C`

Generates `.c` files and places the files in your build folder.

`C++`

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

`C++ (Encapsulated)`

Generates C++ encapsulated `.cpp` files and places the files in your build folder. 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 an 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 an Embedded Coder license.

Tip

You might need to configure the Simulink Coder 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"

"Function Prototype Control"

"C++ Encapsulation Interface Control"

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 Simulink Coder makefile build process to minimize compilation time.

`Optimizations on (faster runs)`
: Customizes compilation during the Simulink Coder makefile build process to minimize run time.

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

Tips

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

Dependencies

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

Command-Line Information

Parameter: `RTWCompilerOptimization`
Type: string
Value: `'Off'` | `'On'` | `'Custom'`
Default: `'Off'`

Recommended Settings

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

See Also

- “Custom compiler optimization flags” on page 6-12
- “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" on page 6-10.

See Also

- "Compiler optimization level" on page 6-10
- "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

- “Specifying 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 MATLAB command, invoked when you start a build, controls the Simulink Coder 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 MATLAB language 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 a MATLAB language file.
- You can customize your build process by modifying an existing template makefile or by providing your own template makefile.

Dependency

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

Command-Line Information

Parameter: `TemplateMakefile`
Type: string
Value: 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 an Embedded Coder license when generating code.

Command-Line Information

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

Recommended Settings

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

See Also

“Custom Storage Classes”

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 an Embedded Coder license when generating code.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	Off
Traceability	No impact

Application	Setting
Efficiency	On
Safety precaution	No impact

See Also

- "Signals with Test Points" in the Simulink Coder User's Guide
- "Working with Test Points" in the Simulink User's Guide
- "Signals" in the Simulink Coder 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.

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

Tips

For more objectives, specify an ERT-based target.

Dependency

These parameters appear only for GRT-based targets.

Command-Line Information

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

Recommended Settings

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

See Also

- "Application Considerations" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder 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 an Embedded Coder license when generating code.

Command-Line Information

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

See Also

- "Application Considerations" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder User's Guide.

Set objectives

Opens Configuration Set Objectives dialog box.

Dependency

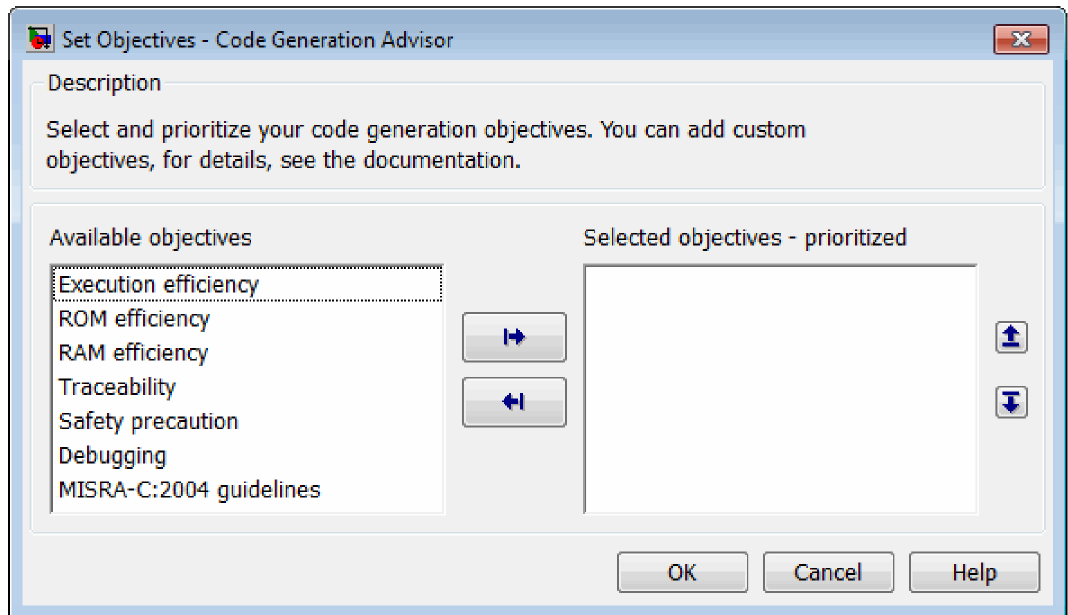
This button appears only for ERT-based targets.

See Also

- "Application Considerations" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder User's Guide.

Set Objectives — Code Generation Advisor Dialog Box

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



Settings

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

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

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

Objectives. List of available objectives.

`Execution efficiency` — Configure code generation settings to achieve fast execution time.

`ROM efficiency` — Configure code generation settings to reduce ROM usage.

`RAM efficiency` — Configure code generation settings to reduce RAM usage.

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

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

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

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

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

- The model configuration settings for compliance with the MISRA-C:2004 configuration setting recommendations.
- For blocks that are not supported or recommended for MISRA-C:2004 compliant code generation.

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

Dependency

This dialog box appears only for ERT-based targets.

Command-Line Information

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

Default: `{''}`

See Also

- "Application Considerations" in the Embedded Coder documentation.
- "Application Objectives" in the Simulink Coder 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 and provide suggestions for changing your model to meet the objectives.

Dependency

You must specify objectives before checking the model.

See Also

- “Application Considerations” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder 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 Simulink Coder software continues generating code. The code generation report summary and file headers indicate the specified objectives and the validation result.

`On (stop for warnings)`
: Checks whether the model meets code generation objectives using the Code Generation Objectives checks in the Code Generation Advisor. If the Code Generation Advisor reports a warning, the Simulink Coder software does not continue generating code.

Command-Line Information

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

Recommended Settings

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

See Also

- “Application Considerations” in the Embedded Coder documentation.
- “Application Objectives” in the Simulink Coder 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"

Code Generation Pane: Report

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

Create code generation report   Launch report automatically

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

Navigation
Code-to-model
Model-to-code   Configure...
Traceability Report Contents
Eliminated / virtual blocks
Traceable Simulink blocks
Traceable Stateflow objects
Traceable MATLAB functions
Metrics
Static code metrics

In this section...

"Code Generation: Report Tab Overview" on page 6-40

"Create code generation report" on page 6-41

"Launch report automatically" on page 6-44

"Code-to-model" on page 6-46

"Model-to-code" on page 6-48

In this section...

"Configure" on page 6-50

"Eliminated / virtual blocks" on page 6-51

"Traceable Simulink blocks" on page 6-53

"Traceable Stateflow objects" on page 6-55

"Traceable MATLAB functions" on page 6-57

"Static code metrics" on page 6-59

Code Generation: Report Tab Overview

Control the Code Generation report that the Simulink Coder software automatically creates.

Configuration

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

See Also

- "Generating a Report"

  If you have an Embedded Coder license, see also "Report Generation".
- "Code Generation Pane: Report" on page 6-38

Create code generation report

Document generated code in an HTML report.

Settings

Default: Off

On

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

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

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

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

- If you selected the traceability option **Model-to-code**, you can view the generated code for any block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **Code Generation > Navigate to Code** (requires an Embedded Coder license and the ERT target).
- If you set the **Code coverage tool** parameter on the **Code Generation > SIL and PIL Verification** pane, you can view the code coverage data and annotations in the generated code in the HTML Code Generation Report (requires an Embedded Coder license and the ERT target).

Off
Does not generate a summary of files.

Dependency

This parameter enables and selects

- "Launch report automatically" on page 6-44
- "Code-to-model" on page 6-46 (ERT target)

This parameter enables

- "Model-to-code" on page 6-48 (ERT target)
- "Eliminated / virtual blocks" on page 6-51 (ERT target)
- "Traceable Simulink blocks" on page 6-53 (ERT target)
- "Traceable Stateflow objects" on page 6-55 (ERT target)
- "Traceable MATLAB functions" on page 6-57 (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

“Generating a Report”

“Reviewing Generated Code”

If you have an Embedded Coder license, see also “Report Generation”.

If you have an Embedded Coder license, see also “Using a Code Coverage Tool in SIL Simulation”.

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` folder.

Dependency

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

Command-Line Information

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

Recommended Settings

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

See Also

"Generating a Report"

"Reviewing Generated Code"

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

Code-to-model

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

Settings

Default: Off

On

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

Off

Omits hyperlinks from the generated report.

Tip

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled and selected by **Create code generation report**.
- You must select **Include comments** on the **Code Generation > Comments** 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

"Report Generation"

Model-to-code

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

Settings

Default: Off

On

Includes model-to-code highlighting support in the Code Generation report. To highlight the generated code for a Simulink block, Stateflow object, or MATLAB script in the Code Generation report, right-click the item and select **Code Generation > 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 an Embedded Coder license when generating code.
- This parameter is enabled when you select **Create code generation report**.
- This parameter selects:
  - "Eliminated / virtual blocks" on page 6-51
  - "Traceable Simulink blocks" on page 6-53
  - "Traceable Stateflow objects" on page 6-55
  - "Traceable MATLAB functions" on page 6-57
- You must select the following parameters to use this parameter:

- "Include comments" on page 6-64 on the **Code Generation** > **Comments** tab
- At least one of the following:
  - "Eliminated / virtual blocks" on page 6-51
  - "Traceable Simulink blocks" on page 6-53
  - "Traceable Stateflow objects" on page 6-55
  - "Traceable MATLAB functions" on page 6-57

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

"Report Generation"

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 folder containing previously-generated model code to highlight. Applying your build folder selection will attempt to load traceability information from the earlier build, for which **Model-to-code** must have been selected.

Dependency

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is enabled by "Model-to-code" on page 6-48.

See Also

"Report Generation"

Eliminated / virtual blocks

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

Settings

Default: Off

☑ On
Includes a summary of eliminated and virtual blocks in the Code Generation report.

☐ Off
Does not include a summary of eliminated and virtual blocks.

Dependencies

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

Command-Line Information

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

Recommended Settings

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

See Also

“Report Generation”

Traceable Simulink blocks

Include summary of Simulink blocks in Code Generation report.

Settings

Default: Off

On

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

Off

Does not include a summary of Simulink blocks.

Dependencies

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

Command-Line Information

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

Recommended Settings

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

See Also

"Report Generation"

Traceable Stateflow objects

Include summary of Stateflow objects in Code Generation report.

Settings

Default: Off

On

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

Off

Does not include a summary of Stateflow objects.

Dependencies

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

Command-Line Information

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

Recommended Settings

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

See Also

"Report Generation"

"Traceability of Stateflow Objects in Generated Code"

Traceable MATLAB functions

Include summary of MATLAB functions in Code Generation report.

Settings

Default: Off

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

☐ Off
Does not include a summary of MATLAB functions.

Dependencies

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

Command-Line Information

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

Recommended Settings

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

See Also

“Report Generation”

Static code metrics

Include static code metrics report in the Code Generation report.

Settings

Default: Off

☑ On
Include static code metrics report in the Code Generation report. The static code metrics report does not support the C++ target language.

☐ Off
Omit static code metrics report from the Code Generation report.

Dependencies

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

Command-Line Information

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

Recommended Settings

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

See Also

"Analyze Static Code Metrics of the Generated Code"

Code Generation Pane: Comments

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

Overall control

- [x] Include comments

Auto generated comments

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

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

Custom comments

- [ ] Simulink block descriptions
- [ ] Simulink data object descriptions
- [ ] Custom comments (MPT objects only)
- [ ] Stateflow object descriptions
- [ ] Requirements in block comments
- [ ] MATLAB function help text

In this section...

"Code Generation: Comments Tab Overview" on page 6-63
"Include comments" on page 6-64
"Simulink block / Stateflow object comments" on page 6-66
"MATLAB source code as comments" on page 6-67
"Show eliminated blocks" on page 6-69

In this section...

"Verbose comments for SimulinkGlobal storage class" on page 6-70

"Simulink block descriptions" on page 6-71

"Simulink data object descriptions" on page 6-73

"Custom comments (MPT objects only)" on page 6-75

"Custom comments function" on page 6-77

"Stateflow object descriptions" on page 6-79

"Requirements in block comments" on page 6-81

"MATLAB function help text" on page 6-83

Code Generation: Comments Tab Overview

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

See Also

"Code Generation Pane: Comments" on page 6-61

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

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	On

Simulink block / Stateflow object comments

Specify whether to insert Simulink block and Stateflow object comments.

Settings

Default: on

On

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

Off

Suppresses comments.

Dependency

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

Command-Line Information

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

Recommended Settings

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

MATLAB source code as comments

Specify whether to insert MATLAB source code as comments.

Settings

Default: off

On

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

Includes the function signature in the function banner.

Off

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

Dependency

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

Command-Line Information

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

Recommended Settings

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

See Also

"Including MATLAB Source Code as Comments in the Generated Code"

Show eliminated blocks

Specify whether to insert eliminated block's comments.

Settings

Default: off

On

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

Off

Suppresses statements.

Dependency

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

Command-Line Information

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

Recommended Settings

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

Verbose comments for SimulinkGlobal storage class

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

Settings

Default: off

On

Generates parameter comments regardless of the number of parameters.

Off

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

Dependency

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

Command-Line Information

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

Recommended Settings

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

Simulink block descriptions

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

Settings

Default: off

On

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

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

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

Off

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

Dependency

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

Command-Line Information

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

Recommended Settings

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

See Also

"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 an Embedded Coder license when generating code.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

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 an Embedded Coder license when generating code.
- This parameter requires that you include the comments in a function defined in a MATLAB language file or TLC file that you specify with **Custom comments function**.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

"Adding Custom Comments"

Custom comments function

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

Settings

Default: `''`

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

Tip

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

Dependency

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

Command-Line Information

Parameter: `CustomCommentsFcn`
Type: string
Value: 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 Simulink Coder software includes the requirement descriptions in the generated code in the following locations.

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

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

Off

Suppresses the generation of comments for block requirement descriptions.

Dependency

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

"Requirements Information in Generated Code" in the Simulink Verification and Validation documentation

MATLAB function help text

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

Settings

Default: off

☑ On
Inserts MATLAB function help text in the function banner.

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

Dependency

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

Command-Line Information

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

Recommended Settings

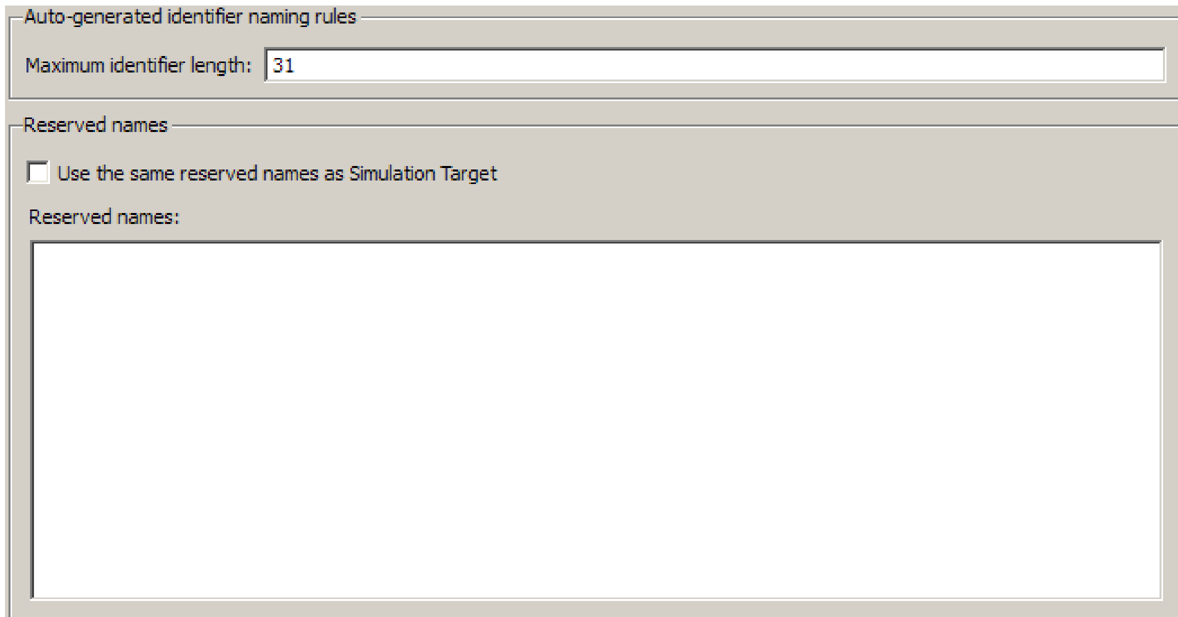
Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

"Including MATLAB Function Help Text in the Function Banner"

Code Generation Pane: Symbols

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



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

Auto-generated identifier naming rules

Identifier format control

Global variables:	RN$M
Global types:	NR$M
Field name of global types:	NM
Subsystem methods:	RNMF
Subsystem method arguments:	rt$I$N$M
Local temporary variables:	NM
Local block output variables:	rtb_NM
Constant macros:	RN$M

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...

"Code Generation: Symbols Tab Overview" on page 6-88
"Global variables" on page 6-89
"Global types" on page 6-91
"Field name of global types" on page 6-94
"Subsystem methods" on page 6-96
"Subsystem method arguments" on page 6-99
"Local temporary variables" on page 6-101
"Local block output variables" on page 6-103
"Constant macros" on page 6-105
"Minimum mangle length" on page 6-107
"Maximum identifier length" on page 6-109
"Generate scalar inlined parameter as" on page 6-111
"Signal naming" on page 6-112
"M-function" on page 6-114
"Parameter naming" on page 6-116
"#define naming" on page 6-118
"Use the same reserved names as Simulation Target" on page 6-120
"Reserved names" on page 6-121

Code Generation: Symbols Tab Overview

Select the automatically generated identifier naming rules.

See Also

- "Configuring Generated Identifiers"
- "Code Generation Pane: Symbols" on page 6-85

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 an 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" in the Embedded Coder documentation
- "Name Mangling" in the Embedded Coder documentation
- "Model Referencing Considerations" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

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 an Embedded Coder license when generating code.

Command-Line Information

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

Recommended Settings

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

See Also

- “Specifying Identifier Formats” in the Embedded Coder documentation

- “Name Mangling” in the Embedded Coder documentation
- “Model Referencing Considerations” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

Field name of global types

Customize generated field names of global types.

Settings

Default: `$N$M`

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

Token	Description
`$A`	Insert data type acronym (for example, `i32` for long integers) into signal and work vector identifiers.
`$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 an 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" in the Embedded Coder documentation
- "Name Mangling" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Subsystem methods

Customize generated function names for reusable subsystems.

Settings

Default: `$R$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
`$F`	Insert method name (for example, `_Update` for update method).
`$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 an 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

Application	Setting
Efficiency	No impact
Safety precaution	`$R$N$M$F`

See Also

- "Specifying Identifier Formats" in the Embedded Coder documentation
- "Name Mangling" in the Embedded Coder documentation
- "Model Referencing Considerations" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Subsystem method arguments

Customize generated function argument names for reusable subsystems.

Settings

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

Token	Description
`$I`	Insert an `u` if the argument is an input. Insert a `y` if the argument is an output. Optional.
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$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 an Embedded Coder license when generating code.

Command-Line Information

Parameter: `CustomSymbolStrFcnArg`
Type: string
Value: any valid combination of tokens
Default: `'rtu_$N$M'` or `'rty_$N$M'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid combinations of tokens
Efficiency	No impact
Safety precaution	`rtu_$N$M` or `rty_$N$M`

See Also

- "Code Generation Pane: Symbols" on page 6-85
- "Specifying Identifier Formats" in the Embedded Coder documentation
- "Name Mangling" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Local temporary variables

Customize generated local temporary variable identifiers.

Settings

Default: `$N$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 an 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" in the Embedded Coder documentation
- "Name Mangling" in the Embedded Coder documentation
- "Model Referencing Considerations" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Local block output variables

Customize generated local block output variable identifiers.

Settings

Default: `rtb_$N$M`

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

Token	Description
`$A`	Insert data type acronym (for example, `i32` for long integers) into signal and work vector identifiers.
`$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 an Embedded Coder license when generating code.

Command-Line Information

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

Recommended Settings

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

See Also

- "Specifying Identifier Formats" in the Embedded Coder documentation
- "Name Mangling" in the Embedded Coder documentation
- "Identifier Format Control Parameters Limitations" in the Embedded Coder documentation

Constant macros

Customize generated constant macro identifiers.

Settings

Default: `$R$N$M`

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

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions. Required.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$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 an 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” in the Embedded Coder documentation
- “Name Mangling” in the Embedded Coder documentation
- “Model Referencing Considerations” in the Embedded Coder documentation
- “Identifier Format Control Parameters Limitations” in the Embedded Coder documentation

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. The maximum possible value is 15. 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; it allows for over 1.5 million collisions for a particular identifier before the mangle length increases.
- Set the value to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

Dependency

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

Command-Line Information

Parameter: `MangleLength`
Type: integer
Value: value between `1` and `15`
Default: `1`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	1
Efficiency	No impact
ty precaution	No impact

See Also

- "Name Mangling" in the Embedded Coder documentation
- "Traceability" in the Embedded Coder documentation
- "Minimizing Name Mangling" in the Embedded Coder documentation

Maximum identifier length

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

Settings

Default: `31`
Minimum: `31`
Maximum: `256`

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

Tips

- Consider increasing identifier length for models having a deep hierarchical structure.
- When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate the root model name and 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
ty precaution	>30

See Also

"Referenced Models" in the Simulink Coder documentation

Generate scalar inlined parameter as

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

Settings

Default: `Literals`

`Literals`

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

`Macros`

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

Dependencies

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

Command-Line Information

Parameter: `InlinedPrmAccess`
Type: string
Value: `'Literals'` | `'Macros'`
Default: `'Literals'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	`Macros`
Efficiency	`Literals`
ty 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 MATLAB function specified with the **M-function** parameter to create identifiers for signal names in the generated code.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Custom M-function` enables **M-function**.
- This parameter must be the same for top-level and referenced models.
- If you give a value to the **Alias** parameter of an `MPT.Signal` or `Simulink.Signal` data object, that value overrides the specification of the **Signal naming** parameter.

Limitation

This parameter has no effect on signal names that are specified by an embedded signal object created using the **Code Generation** tab of a **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
ty precaution	No impact

See Also

- "Applying Naming Rules to Identifiers Globally" in the Embedded Coder documentation
- "Functions and Scripts" in the MATLAB documentation

M-function

Specify rule for naming identifiers in generated code.

Settings

Default: `''`

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

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

Tip

The MATLAB language file must be in the MATLAB path.

Dependencies

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

Command-Line Information

Parameter: `DefineNamingFcn`
Type: string
Value: any MATLAB language file
Default: `''`

Recommended Settings

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

See Also

- “Applying Naming Rules to Identifiers Globally” in the Embedded Coder documentation
- “Functions and Scripts” in the MATLAB documentation

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

Dependencies

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

Command-Line Information

Parameter: `ParamNamingRule`
Type: string
Value: `'None'` | `'UpperCase'` | `'LowerCase'` | `'Custom'`
Default: `'None'`

Recommended Settings

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

See Also

- "Applying Naming Rules to Identifiers Globally" in the Embedded Coder documentation
- "Functions and Scripts" in the MATLAB documentation

#define naming

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

Settings

Default: `None`

`None`
: 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 MATLAB function specified with the **M-function** parameter to create identifiers for `#define` parameter names in the generated code.

Dependencies

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

Command-Line Information

Parameter: `DefineNamingRule`
Type: string
Value: `'None'` | `'UpperCase'` | `'LowerCase'` | `'Custom'`
Default: `'None'`

Recommended Settings

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

See Also

- "Applying Naming Rules to Identifiers Globally" in the Embedded Coder documentation
- "Functions and Scripts" in the MATLAB documentation

Use the same reserved names as Simulation Target

Specify whether to use the same reserved names as those specified in the **Simulation Target > Symbols** pane.

Settings

Default: Off

☑ On
  Enables using the same reserved names as those specified in the **Simulation Target > Symbols** pane.

☐ 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
ty precaution	No impact

Reserved names

Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code.

Settings

Default: `{}`

This action changes the names of variables or functions in the generated code to avoid name conflicts with identifiers in custom code. Reserved names must be shorter than 256 characters.

Tips

- Do not enter Simulink Coder keywords since these names cannot be changed in the generated code. For a list of keywords to avoid, see "Reserved Keywords" in the *Simulink Coder 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
ty precaution	No impact

Code Generation Pane: Custom Code

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

☐ Use the same custom code settings as Simulation Target

Include custom C code in generated:

- Source file
- Header file
- Initialize function
- Terminate function

Source file:

Include list of additional:

- Include directories
- Source files
- Libraries

Include directories:

In this section...

"Code Generation: Custom Code Tab Overview" on page 6-126

"Use the same custom code settings as Simulation Target" on page 6-127

"Use local custom code settings (do not inherit from main model)" on page 6-128

"Source file" on page 6-130

"Header file" on page 6-131

"Initialize function" on page 6-132

"Terminate function" on page 6-133

"Include directories" on page 6-134

"Source files" on page 6-136

"Libraries" on page 6-138

Code Generation: Custom Code Tab Overview

Enter custom code to include in generated model files and create a list of additional folders, source files, and libraries to use when building the model.

Configuration

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

See Also

- "Model Configuration Code Insertion"
- "Code Generation Pane: Custom Code" on page 6-123

Use the same custom code settings as Simulation Target

Specify whether to use the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.

Settings

Default: Off

On

Enables using the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.

Off

Disables using the same custom code settings as those in the **Simulation Target** > **Custom Code** pane.

Command-Line Information

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

Recommended Settings

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

See Also

"Model Configuration Code Insertion"

Use local custom code settings (do not inherit from main model)

Specify if a library model can use custom code settings that are unique from the main model.

Settings

Default: Off

☑ On
: Enables a library model to use custom code settings that are unique from the main model.

☐ Off
: Disables a library model from using custom code settings that are unique from the main model.

Dependency

This parameter is available only for library models that contain MATLAB Function blocks, Stateflow charts, or Truth Table blocks. To access this parameter, select **Tools > Open Code Generation Target** in the MATLAB Function Block 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

See Also

"Model Configuration Code Insertion"

Source file

Specify custom code to include near the top of the generated model source file.

Settings

Default:''

The Simulink Coder 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

See Also

"Model Configuration Code Insertion"

Header file

Specify custom code to include near the top of the generated model header file.

Settings

Default:''

The Simulink Coder 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

See Also

"Model Configuration Code Insertion"

Initialize function

Specify custom code to include in the generated model initialize function.

Settings

Default: `''`

The Simulink Coder software places code inside the model's initialize function in the *model*.c or *model*.cpp file.

Command-Line Information

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

Recommended Settings

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

See Also

"Model Configuration Code Insertion"

Terminate function

Specify custom code to include in the generated model terminate function.

Settings

Default: `''`

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

Dependency

A terminate function is generated only if you select the **Terminate function required** check box on the **Code Generation** 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

See Also

"Model Configuration Code Insertion"

Include directories

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

Settings

Default:`''`

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

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

Note If you specify a Windows path string containing one or more spaces, you must enclose the string in double quotes. For example, the second and third path strings in the **Include directories** entry below must be double-quoted:

```
C:\Project "C:\Custom Files" "C:\Library Files"
```

If you set the equivalent command-line parameter `CustomInclude`, each path string containing spaces must be separately double-quoted within the single-quoted third argument string, for example,

```
>> set_param('mymodel', 'CustomInclude', ...
             'C:\Project "C:\Custom Files" "C:\Library Files"')
```

Command-Line Information

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

Recommended Settings

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

See Also

"Model Configuration Code Insertion"

Source files

Specify a list of additional source files to compile and link with the generated code.

Settings

Default: `''`

Enter a space-separated list of source files to compile and link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

The file name is sufficient if the file is in the current MATLAB folder or in one of the include folders.

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

See Also

"Model Configuration Code Insertion"

Libraries

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

Settings

Default: `''`

Enter a space-separated list of static library files to link with the generated code.

Limitation

This parameter does not support Windows file names that contain embedded spaces.

Tip

The file name is sufficient if the file is in the current MATLAB folder or in one of the include folders.

Command-Line Information

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

Recommended Settings

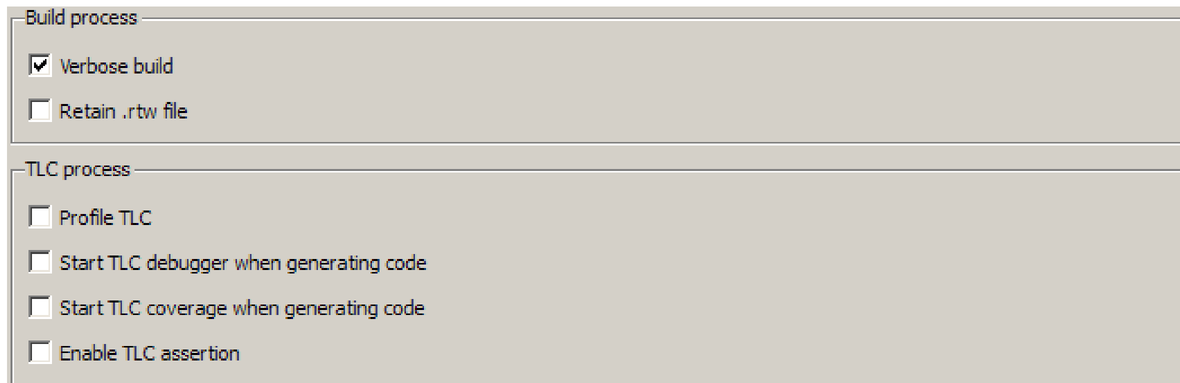
Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Model Configuration Code Insertion"

Code Generation Pane: Debug

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



In this section...

"Code Generation: Debug Tab Overview" on page 6-142

"Verbose build" on page 6-143

"Retain .rtw file" on page 6-144

"Profile TLC" on page 6-145

"Start TLC debugger when generating code" on page 6-146

"Start TLC coverage when generating code" on page 6-148

"Enable TLC assertion" on page 6-149

Code Generation: Debug Tab Overview

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

See Also

- "Debugging"
- "Code Generation Pane: Debug" on page 6-140

Verbose build

Display code generation progress.

Settings

Default: on

On

The MATLAB Command Window displays progress information indicating code generation stages and compiler output during code generation.

Off

Does not display progress information.

Command-Line Information

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

Recommended Settings

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

See Also

"Debugging"

Retain .rtw file

Specify *model*.rtw file retention.

Settings

Default: off

On

Retains the *model*.rtw file in the current build folder. This parameter is useful if you are modifying the target files and need to look at the file.

Off

Deletes the *model*.rtw from the build folder at the end of the build process.

Command-Line Information

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

Recommended Settings

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

See Also

"Debugging"

Profile TLC

Profile the execution time of TLC files.

Settings

Default: off

On

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

Off

Does not profile the performance.

Command-Line Information

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

Recommended Settings

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

See Also

"Debugging"

Start TLC debugger when generating code

Specify use of the TLC debugger

Settings

Default: Off

☑ On
The TLC debugger starts during code generation.

☐ Off
Does not start the TLC debugger.

Tips

- You can also start the TLC debugger by entering the `-dc` argument into the **System target file** field.
- To invoke the debugger and run a debugger script, enter the `-df` *`filename`* argument into the **System target file** field.

Command-Line Information

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

Recommended Settings

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

See Also

“Debugging”

Start TLC coverage when generating code

Generate the TLC execution report.

Settings

Default: off

☑ On

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

☐ Off

Does not generate a report.

Tip

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

Command-Line Information

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

Recommended Settings

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

See Also

"Debugging"

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

See Also

"Debugging"

Code Generation Pane: Interface

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

Software environment

Target function library: C89/C90 (ANSI)

Shared code placement: Auto

☑ Support non-finite numbers

Data exchange

☑ MAT-file logging   MAT-file variable name modifier: rt_

Interface: None

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

Software environment

Target function library: C89/C90 (ANSI) Custom...

Shared code placement: Auto

Support: [x] floating-point numbers [x] non-finite numbers [x] complex numbers

[x] absolute time [ ] continuous time [ ] non-inlined S-functions

[ ] variable-size signals

Multiword type definitions: System defined

Code interface

[ ] GRT compatible call interface [x] Single output/update function [x] Terminate function required

[ ] Generate reusable code

Generate preprocessor conditionals: Use local settings

[ ] Suppress error status in real-time model data structure [ ] Combine signal/state structures

Configure Model Functions

Data exchange

[ ] MAT-file logging

Interface: None

In this section...

"Code Generation: Interface Tab Overview" on page 6-154
"Target function library" on page 6-155
"Custom" on page 6-158
"Shared code placement" on page 6-159
"Support: floating-point numbers" on page 6-161
"Support: non-finite numbers" on page 6-163
"Support: complex numbers" on page 6-165
"Support: absolute time" on page 6-166
"Support: continuous time" on page 6-168
"Support: non-inlined S-functions" on page 6-170
"Support: variable-size signals" on page 6-172
"Multiword type definitions" on page 6-173
"Maximum word length" on page 6-175
"GRT compatible call interface" on page 6-177
"Single output/update function" on page 6-179
"Terminate function required" on page 6-181
"Generate reusable code" on page 6-183
"Reusable code error diagnostic" on page 6-186
"Pass root-level I/O as" on page 6-188
"Block parameter visibility" on page 6-190
"Internal data visibility" on page 6-192
"Block parameter access" on page 6-194
"Internal data access" on page 6-196
"External I/O access" on page 6-198
"Generate destructor" on page 6-200
"Use operator new for referenced model object registration" on page 6-202

In this section...

"Generate preprocessor conditionals" on page 6-204

"Suppress error status in real-time model data structure" on page 6-206

"Combine signal/state structures" on page 6-208

"Configure Model Functions" on page 6-211

"Configure C++ Encapsulation Interface" on page 6-212

"MAT-file logging" on page 6-213

"MAT-file variable name modifier" on page 6-216

"Interface" on page 6-218

"Generate C API for: signals" on page 6-221

"Generate C API for: parameters" on page 6-222

"Generate C API for: states" on page 6-223

"Generate C API for: root-level I/O" on page 6-224

"Transport layer" on page 6-225

"MEX-file arguments" on page 6-227

"Static memory allocation" on page 6-229

"Static memory buffer size" on page 6-231

Code Generation: Interface Tab Overview

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

See Also

- “Specifying Target Interfaces”
- “Code Generation Pane: Interface” on page 6-150

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`.

`C++ (ISO)`
: Generates calls to the ISO/IEC 14882:2003 C++ standard math library.

`Intel IPP (ANSI)`
: Generates calls to the Intel Performance Primitives (IPP) ANSI® library.

`Intel IPP (ISO)`
: Generates calls to the Intel Performance Primitives (IPP) ISO library.

`Intel IPP/SSE (GNU)`
: Generates calls to the GNU libraries for Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE).

Note

- Additional values might be listed for licensed target products, for Embedded Targets and Desktop Targets, or if you have created and registered target function libraries using the Embedded Coder product.
- The list of **Target function library** values is filtered based on the **Device vendor** value selected for your model on the **Hardware Implementation** pane. If you set **Device vendor** to `Generic`, the list of **Target function library** values shows all registered TFLs.

Tip

Before setting this parameter, verify that your compiler supports the library you want to use. If you select a parameter value that your compiler does not support, compiler errors can occur.

Dependencies

The `C++ (ISO)` math library is available for use only if you select a compatible value for the **Language** parameter on the **Code Generation** pane of the Configuration Parameters dialog box:

- For the GRT target, select `C++`.
- For an ERT-based target, select `C++` or `C++ (Encapsulated)`.

Using an ERT-based target and the `C++ (Encapsulated)` value for code generation requires an Embedded Coder license.

Command-Line Information

Parameter: `TargetFunctionLibrary`
Type: string
Value: `'ANSI_C' | 'C99 (ISO)' | 'GNU99 (GNU)' | 'C++ (ISO)'`
Default: `'ANSI_C'`

Recommended Settings

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

See Also

"Specifying Target Interfaces"

Custom

Click the **Custom** button to open the Code Replacement Tool. With this tool, you can you create and manage the code replacement tables that make up a target function library (TFL).

Dependencies

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

See Also

- "Creating and Managing Code Replacement Tables Using the Code Replacement Tool"
- Code Replacement

Shared code placement

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Settings

Default: `Auto`

`Auto`
: Operates as follows:

  - When the model contains Model blocks, places utility code within the `slprj/target/_sharedutils` folder.
  - When the model does not contain Model blocks, places utility code in the build folder (generally, in *`model`*`.c` or *`model`*`.cpp`).

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

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	`Shared location` (GRT) No impact (ERT)
Traceability	`Shared location` (GRT) No impact (ERT)
Efficiency	No impact (execution, RAM) `Shared location` (ROM)
Safety precaution	No impact

See Also

- "Specifying Target Interfaces"
- "Shared Utility Code"

Support: floating-point numbers

Specify whether to generate floating-point data and operations.

Settings

Default: On (GUI), `'off'` (command-line)

On

Generates floating-point data and operations.

Off

Generates pure integer code. If you clear this option, an error occurs if the code generator encounters floating-point data or expressions. The error message reports offending blocks and parameters.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an 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: non-finite numbers

Specify whether to generate nonfinite data and operations on nonfinite data.

Settings

Default: on

On

Generates nonfinite data (for example, `NaN` and `Inf`) and related operations.

Off

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

Note Code generation is optimized with the assumption that there is no nonfinite data. However, if your application produces nonfinite numbers through signal data or MATLAB code, the behavior of the generated code might be inconsistent with simulation results when processing nonfinite data.

Dependencies

- For ERT-based targets, this parameter is enabled by **Support: floating-point numbers**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (execution, ROM), No impact (RAM)
Safety precaution	Off

Support: complex numbers

Specify whether to generate complex data and operations.

Settings

Default: on

On

Generates complex numbers and related operations.

Off

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

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Off (for real only)
Safety precaution	No impact

Support: absolute time

Specify whether to generate and maintain integer counters for absolute and elapsed time values.

Settings

Default: on

On

Generates and maintains integer counters for blocks that require absolute or elapsed time values. Absolute time is the time from the start of program execution to the present time. An example of elapsed time is time elapsed between two trigger events.

If you select this option and the model does not include blocks that use time values, the target does not generate the counters.

Off

Does not generate integer counters to represent absolute or elapsed time values. If you do not select this option and the model includes blocks that require absolute or elapsed time values, an error occurs during code generation.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- You must select this parameter if your model includes blocks that require absolute or elapsed time values.

Command-Line Information

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

Recommended Settings

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

See Also

"Using Timers"

Support: continuous time

Specify whether to generate code for blocks that use continuous time.

Settings

Default: off

On

Generates code for blocks that use continuous time.

Off

Does not generate code for blocks that use continuous time. If you do not select this option and the model includes blocks that use continuous time, an error occurs during code generation.

Dependencies

- This option only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This option must be on if your model includes blocks that require absolute or elapsed time values.
- This option must be off when generating an S-function wrapper for an ERT-based 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 (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

- "Using Discrete and Continuous Time"
- "Generating S-Function Wrappers"

Support: non-inlined S-functions

Specify whether to generate code for noninlined S-functions.

Settings

Default: Off

☑ On

Generates code for noninlined S-functions.

☐ Off

Does not generate code for noninlined S-functions. If this parameter is off and the model includes a noninlined S-function, an error occurs during the build process.

Tip

- Inlining S-functions is highly advantageous in production code generation, for example, for implementing device drivers. In such cases, clear this option to enforce use of inlined S-functions for code generation.
- Noninlined S-functions require additional memory and computation resources, and can result in significant performance issues. Consider using an inlined S-function when efficiency is a concern.

Dependencies

- This parameter only appears for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- Selecting this parameter also selects **Support: floating-point numbers** and **Support: non-finite numbers**. If you clear **Support: floating-point numbers** or **Support: non-finite numbers**, a warning is displayed during code generation because these parameters are required by the S-function interface.

Command-Line Information

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

Recommended Settings

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

See Also

- "Generating S-Function Wrappers"
- ""

Support: variable-size signals

Specify whether to generate code for models that use variable-size signals.

Settings

Default: Off

☑ On
Generates code for models that use variable-size signals.

☐ Off
Does not generate code for models that use variable-size signals. If this parameter is off and the model uses variable-size signals, an error occurs during code generation.

Dependencies

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

Command-Line Information

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

Recommended Settings

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

Multiword type definitions

Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.

Settings

Default: `System defined`

`System defined`
: Use the default system type definitions for multiword data types in generated code. During code generation, if multiword usage is detected, multiword 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 an Embedded Coder license when generating code.
- Selecting the value `User defined` for this parameter enables the associated parameter **Maximum word length**.

Command-Line Information

Parameter: ERTMultiwordTypeDef
Type: string
Value: `'System defined'` | `'User defined'`
Default: `'System defined'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	Specifying `User defined` and a low value for **Maximum word length** reduces the size of the generated file `rtwtypes.h`
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 an Embedded Coder license when generating code.
- This parameter is enabled by selecting the value `User defined` for the parameter **Multiword type definitions**.

Command-Line Information

Parameter: 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 `rtwtypes.h`
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 an 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 (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

"Using Discrete and Continuous 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 into a single function, and instead generates the code in separate `model_output` and `model_update` functions.

Tips

Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. See "Model Blocks and Direct Feedthrough" for details.

Simulink Coder ignores this parameter for a referenced model if any of the following conditions apply to that model:

- Is multi-rate
- Has a continuous sample time
- Is logging states (using the **States** or **Final States** parameters in the **Configuration Parameters > Data Import/Export** pane

Dependencies

- This option only appears for ERT-based targets with **Language** set to `C` or `C++` (not `C++ (Encapsulated)`).
- This parameter requires an 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 and Scheduling Considerations"

Terminate function required

Specify whether to generate the `model_terminate` function.

Settings

Default: on

☑ On
: Generates a `model_terminate` function. This function contains 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 an 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
Efficiency	Off (execution, ROM), No impact (RAM)
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 the other 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 an 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 that is used by multiple instances of the subsystem and has different sample times, data types, complexity, frame status, or dimensions across the instances
    - Has output marked as a global signal
    - For each instance contains identical blocks with different names or parameter settings
- This parameter has no effect on code generated for function-call subsystems.

Command-Line Information

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

Recommended Settings

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

See Also

- “Entry Point Functions and Scheduling”
- “Subsystems”
- “Code Reuse Limitations”
- “Determining Why Subsystem Code Is Not Reused”
- “Writing S-Functions That Support Code Reuse”
- “Static Main Program Module”
- “Function Prototype Control”
- “Atomic Subsystem Code”
- “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 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 an 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

- "Entry Point Functions and Scheduling"
- "Subsystems"
- "Code Reuse Limitations"
- "Determining Why Subsystem Code Is Not Reused"
- "Atomic Subsystem Code"

Pass root-level I/O as

Control how root-level model input and output are passed to the *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 an 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
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

- "Entry Point Functions and Scheduling"
- "Subsystems"
- "Atomic Subsystem Code"
- `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 an Embedded Coder license when generating code.

Command-Line Information

Parameter: `ParameterMemberVisibility`
Type: string
Value: `'public'` | `'private'` | `'protected'`
Default: `'private'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
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 an Embedded Coder license when generating code.

Command-Line Information

Parameter: `InternalMemberVisibility`
Type: string
Value: `'public' | 'private' | 'protected'`
Default: `'private'`

Recommended Settings

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

See Also

"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 an Embedded Coder license when generating code.

Command-Line Information

Parameter: `GenerateParameterAccessMethods`
Type: string
Value: `'None'` | `'Method'` | `'Inlined method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`

Application	Setting
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 an Embedded Coder license when generating code.

Command-Line Information

Parameter: `GenerateInternalMemberAccessMethods`
Type: string
Value: `'None'` | `'Method'` | `'Inlined method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`
Efficiency	`Inlined method`
Safety precaution	`None`

See Also

"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. The software generates set and get methods as needed for each signal.

`Inlined method`
: Generates inlined access methods for root-level I/O signals for the C++ model class. The software generates set and get methods as needed for each signal.

`Structure-based method`
: Generates noninlined, structure-based access methods for root-level I/O signals for the C++ model class. The software generates one set method, taking the address of the external input structure as an argument, and for external outputs (if applicable), one get method, returning the reference to the external output structure.

`Inlined structure-based method`
: Generates inlined, structure-based access methods for root-level I/O signals for the C++ model class. The software generates one set method, taking the address of the external input structure as an argument,

and for external outputs (if applicable), one get method, returning the reference to the external output structure.

Dependencies

- This parameter appears only for ERT-based targets with **Language** set to `C++ (Encapsulated)`.
- This parameter requires an Embedded Coder license when generating code.

Command-Line Information

Parameter: `GenerateExternalIOAccessMethods`
Type: string
Value: `'None'` | `'Method'` | `'Inlined method'` | `'Structure-based method'` | `'Inlined structure-based method'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	`Inlined method`
Traceability	`Inlined method`
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 an Embedded Coder license when generating code.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
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 an Embedded Coder license when generating code.

Command-Line Information

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

Recommended Settings

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

See Also

"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 > Signals and Parameters** pane of the Configuration Parameters dialog box.
- Deselect the "Ignore custom storage classes" on page 6-21 parameter on the **Code Generation** pane of the Configuration Parameters dialog box.

Dependencies

- This parameter only appears for ERT-based targets.

- This parameter requires an Embedded Coder license when generating code.
- Setting this parameter to `Use local settings` enables **Generate preprocessor conditionals** parameter on the Model block parameters dialog.
- Setting this parameter to `Enable all` or `Disable all` disables the **Generate preprocessor conditionals** check box on the Model block parameters dialog.
- Setting this parameter to `Enable all` sets the **Selected variant** control on the Model block parameter dialog to `(derive from conditions)`.

Command-Line Information

Parameter: `GeneratePreprocessorConditionals`
Type: string
Value: `'Use local settings'` | `'Enable all'` | `'Disable all'`
Default: `'Use local settings'`

Recommended Settings

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

See Also

- "Modeling Variant Systems"
- "Variant Systems"

Suppress error status in real-time model data structure

Specify whether to log or monitor error status.

Settings

Default: off

On

Omits the error status field from the generated real-time model data structure `rtModel`. This option reduces memory usage.

Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

Off

Includes an error status field in the generated real-time model data structure `rtModel`. You can use available macros to monitor the field for error message data or set it with error message data.

Dependencies

- This parameter appears only for ERT-based targets.
- This parameter requires an Embedded Coder license when generating code.
- This parameter is cleared if you select the incompatible option **MAT-file logging**. If you subsequently select this parameter, code generation displays an error.
- Selecting this parameter clears **Support: continuous time**.
- If your application contains multiple integrated models, the setting of this option must be the same for all of the models to avoid unexpected application behavior. For example, if you select the option for one model but not another, an error status might not get registered by the integrated application.

Command-Line Information

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

Recommended Settings

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

See Also

“Using the Real-Time Model Data Structure”

Combine signal/state structures

Specify whether to combine global block signals and global state data into one data structure in the generated code

Settings

Default: Off

On

Combine global block signal data (block I/O) and global state data (DWork vectors) into one data structure in the generated code.

Off

Store global block signals and global states in separate data structures, block I/O and DWork vectors, in the generated code.

Tips

The benefits to setting this parameter to `On` are:

- Enables tighter memory representation through fewer bitfields, which reduces RAM usage
- Enables better alignment of data structure elements, which reduces RAM usage
- Reduces the number of arguments to reusable subsystem and model reference block functions, which reduces stack usage
- Better readable data structures with more consistent element sorting

Example. For a model that generates the following code:

```
/* Block signals (auto storage) */
typedef struct {
  struct {
    uint_T LogicalOperator:1;
    uint_T UnitDelay1:1;
  } bitsForTID0;
} BlockIO;
/* Block states (auto storage) */
typedef struct {
```

```
    struct {
      uint_T UnitDelay_DSTATE:1
      uint_T UnitDelay1_DSTATE:1
    } bitsForTID0;
  } D_Work;
```

If you select **Combine signal/state structures**, the generated code now looks like this:

```
/* Block signals and states (auto storage)
   for system */
typedef struct {
  struct {
    uint_T LogicalOperator:1;
    uint_T UnitDelay1:1;
    uint_T UnitDelay_DSTATE:1;
    uint_T UnitDelay1_DSTATE:1;
  } bitsForTID0;
} D_Work;
```

Dependencies

This parameter:

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

Command-Line Information

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

Recommended Settings

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

See Also

- “The Global Block I/O Structure”
- “State Storage”

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.

Dependencies

- This button appears only for ERT-based targets with **Language** set to `C` or `C++` (not `C++ (Encapsulated)`).
- This button requires an Embedded Coder license when generating code.
- This button is active only if your model uses an attached configuration set. If your model uses a referenced configuration set, the button is greyed out. If you want to configure a model-specific step function prototype for a referenced configuration set, use the MATLAB function prototype control functions described in "Configuring Function Prototypes Programmatically".

See Also

- "Function Prototype Control"
- `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.

Dependencies

- This button appears only for ERT-based targets with **Language** set to `C++ (Encapsulated)`.
- This button requires an Embedded Coder license when generating code.
- This button is active only if your model uses an attached configuration set. If your model uses a referenced configuration set, the button is greyed out. If you want to configure a model-specific C++ encapsulation interface for a referenced configuration set, use the MATLAB C++ encapsulation interface control functions described in "Configuring C++ Encapsulation Interfaces Programmatically".

See Also

- "C++ Encapsulation Interface Control"
- `model_step`
- "Configuring the Step Method for Your Model Class"

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: on for the GRT target, off for ERT-based targets

On

Enables MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in any of the following ways:

- **Configuration Parameters > Data Import/Export**, **Save to workspace** subpane (see "Data Import/Export Pane")
- To Workspace blocks
- Scope blocks with the **Save data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in "Exporting Simulation Data" and "Logging Signals with Scope Blocks". 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

- For the GRT target, selecting this parameter also selects the required option **Support non-finite numbers**. If you subsequently clear **Support non-finite numbers**, an error is displayed during code generation.
- For ERT-based targets, selecting this 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.
- For ERT-based targets, 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**.
- For ERT-based targets, 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: `'on'` for the GRT target, `'off'` for ERT-based targets

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact

Application	Setting
Efficiency	Off
Safety precaution	Off

See Also

- "Logging"
- "Logging Data for Analysis"
- "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

For the GRT target or ERT-based targets, this parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: `LogVarNameModifier`
Type: string
Value: `'none'` | `'rt_'` | `'_rt'`
Default: `'rt_'`

Recommended Settings

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

See Also

- "Logging"
- "Logging Data for Analysis"

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:

- **Generate C API for: signals**
- **Generate C API for: parameters**
- **Generate C API for: states**
- **Generate C API for: root-level I/O**

Selecting **External mode** enables the following parameters:

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

Command-Line Information

Parameter: see table
Type: string
Value: 'on' | 'off'
Default: 'off'

To enable...	Set this parameter...	To this value...
None	RTWCAPIParams, RTWCAPISignals, RTWCAPIStates, RTWCAPIRootIO, ExtMode, GenerateASAP2	'off'
C API	RTWCAPIParams, RTWCAPISignals, RTWCAPIStates RTWCAPIRootIO	'on'
External mode	ExtMode	'on'
ASAP2	GenerateASAP2	'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact during development None for production code generation

See Also

- "Data Interchange Using the C API"
- "Host/Target Communication "

-
- “ASAP2 Data Measurement and Calibration”

Generate C API for: signals

Generate a C API signals structure.

Settings

Default: on

On

Generates C API interface to global block outputs.

Off

Does not generate C API signals.

Dependency

This parameter is enabled by selecting **Interface** > `C API`.

Command-Line Information

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

Recommended Settings

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

See Also

"Data Interchange Using the C API"

Generate C API for: parameters

Generate C API parameter tuning structures.

Settings

Default: on

On

Generates C API interface to global block parameters.

Off

Does not generate C API parameters.

Dependency

This parameter is enabled by selecting **Interface** > `C API`.

Command-Line Information

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

Recommended Settings

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

See Also

"Data Interchange Using the C API"

Generate C API for: states

Generate a C API states structure.

Settings

Default: off

On

Generates C API interface to discrete and continuous states.

Off

Does not generate C API states.

Dependency

This parameter is enabled by selecting **Interface** > `C API`.

Command-Line Information

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

Recommended Settings

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

See Also

"Data Interchange Using the C API"

Generate C API for: root-level I/O

Generate a C API root-level I/O structure.

Settings

Default: off

☑ On
    Generates a C API interface to root-level inputs and outputs.

☐ Off
    Does not generate a C API interface to root-level inputs and outputs.

Dependency

This parameter is enabled by selecting **Interface** > `C API`.

Command-Line Information

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

Recommended Settings

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

See Also

"Data Interchange Using the C API"

Transport layer

Specify the transport protocol for 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 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
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- “Target Interfacing”
- “Creating a TCP/IP Transport Layer for External Communication”

MEX-file arguments

Specify arguments to pass to an external mode interface MEX-file for communicating with executing targets.

Settings

Default: `''`

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

- Network name of your target (for example, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

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

- Verbosity level (`0` for no information or `1` for detailed information)
- Serial port ID (for example, `1` for `COM1`, and so on)
- Baud rate (selected from the set `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600`, `115200`, with a default baud rate of `57600`)

Dependency

Depending on the specified "System target file" on page 6-6, 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”
- “Choosing Communications Protocol for Client and Server Interfaces”

Static memory allocation

Control memory buffer for external mode communication.

Settings

Default: off

On

Enables the **Static memory buffer size** parameter for allocating dynamic memory.

Off

Uses a static memory buffer for external mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- Depending on the specified "System target file" on page 6-6, this parameter is enabled by **Data Exchange > Interface > External mode** or by **External Mode**.
- This parameter enables **Static memory buffer size**.

Command-Line Information

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

Recommended Settings

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

See Also

"External Mode Interface Options"

Static memory buffer size

Specify the memory buffer size for external mode communication.

Settings

Default: `1000000`

Enter the number of bytes to preallocate for external mode communications buffers in the target.

Tips

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

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information

Parameter: `ExtModeStaticAllocSize`
Type: integer
Value: any valid value
Default: `1000000`

Recommended Settings

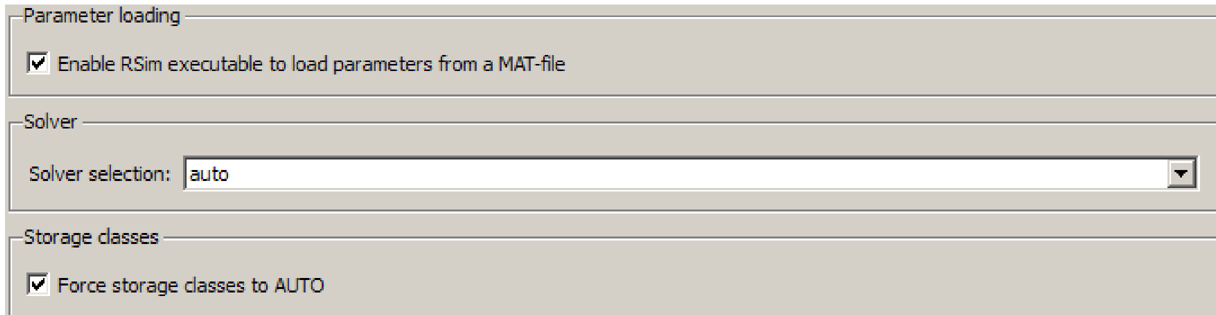
Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"External Mode Interface Options"

Code Generation Pane: RSim Target

The Code Generation RSim Target pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `rsim.tlc` system target file.



In this section...
“Code Generation: RSim Target Tab Overview” on page 6-235
“Enable RSim executable to load parameters from a MAT-file” on page 6-236
“Solver selection” on page 6-237
“Force storage classes to AUTO” on page 6-238

Code Generation: RSim Target Tab Overview

Set configuration parameters for rapid simulation.

Configuration

This tab appears only if you specify `rsim.tlc` as the "System target file" on page 6-6.

See Also

- "Configuring and Building a Model for Rapid Simulation"
- "Running Rapid Simulations"
- "Code Generation Pane: RSim Target" on page 6-233

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 Parameter Structure"

Solver selection

Instruct the target how to select the solver.

Settings

Default: `auto`

`auto`
: Lets the target choose the solver. The target uses the Simulink solver module if you specify a variable-step solver on the Solver pane. Otherwise, the target uses a Simulink Coder built-in solver.

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

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

Command-Line Information

Parameter: `RSIM_SOLVER_SELECTION`
Type: string
Value: `'auto'` | `'usesolvermodule'` | `'usefixstep'`
Default: `'auto'`

Recommended Settings

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

Force storage classes to AUTO

Specify whether to retain your storage class settings in a model or to use the automatic settings.

Settings

Default: on

On

Forces the Simulink software to determine 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

Code Generation Pane: S-Function Target

The Code Generation S-Function Target pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `rtwsfcn.tlc` system target file.

- [x] Create new model
- [ ] Use value for tunable parameters
- [ ] Include custom source code

In this section...

"Code Generation S-Function Target Tab Overview" on page 6-241

"Create new model" on page 6-242

"Use value for tunable parameters" on page 6-243

"Include custom source code" on page 6-244

Code Generation S-Function Target Tab Overview

Control code generated for the S-function target (`rtwsfcn.tlc`).

Configuration

This tab appears only if you specify the S-function target (`rtwsfcn.tlc`) as the "System target file" on page 6-6.

See Also

- "Generated S-Function Block"
- "Code Generation Pane: S-Function Target" on page 6-239

Create new model

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

Settings

Default: on

On

Creates a new model, separate from the current model, containing the generated S-function block.

Off

Generates code but a new model is not created.

Command-Line Information

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

See Also

"Generated S-Function Block"

Use value for tunable parameters

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

Settings

Default: off

On

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

Off

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

Command-Line Information

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

See Also

"Generated S-Function Block"

Include custom source code

Include custom source code in the code generated for the S-function.

Settings

Default: off

On

Always include provided custom source code in the code generated for the S-function.

Off

Do not include custom source code in the code generated for the S-function.

Command-Line Information

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

See Also

"Generated S-Function Block"

Code Generation Pane: Tornado Target

The Code Generation Tornado Target pane includes the following parameters when the Simulink Coder product is installed on your system and you specify the `tornado.tlc` system target file.

Software environment

Target function library: C89/C90 (ANSI)

Shared code placement: Auto

Tornado

MAT-file logging

Code Format RealTime

StethoScope

Download to VxWorks target

VxWorks

Base task priority 30

Task stack size 16384

External mode options

External mode

In this section...

"Code Generation: Tornado Target Tab Overview" on page 6-247
"Target function library" on page 6-248
"Shared code placement" on page 6-250
"MAT-file logging" on page 6-252
"MAT-file variable name modifier" on page 6-254
"Code Format" on page 6-256
"StethoScope" on page 6-257
"Download to VxWorks target" on page 6-259
"Base task priority" on page 6-261
"Task stack size" on page 6-263
"External mode" on page 6-264
"Transport layer" on page 6-266
"MEX-file arguments" on page 6-268
"Static memory allocation" on page 6-270
"Static memory buffer size" on page 6-272

Code Generation: Tornado Target Tab Overview

Control Simulink Coder generated code for the Tornado® target.

Configuration

This tab appears only if you specify `tornado.tlc` as the “System target file” on page 6-6.

See Also

- *Tornado User’s Guide* from Wind River® Systems
- *StethoScope User’s Guide* from Wind River Systems
-
- “Code Generation Pane: Tornado Target” on page 6-245

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`.

`C++ (ISO)`
: Generates calls to the ISO/IEC 14882:2003 C++ standard math library. This setting is visible only if you selected `C++` for the **Language** parameter on the **Code Generation** pane of the Configuration Parameters dialog box.

Note

- Additional values might be listed for Desktop Targets.
- The list of **Target function library** values is filtered based on the **Device vendor** value selected for your model on the **Hardware Implementation** pane. If you set **Device vendor** to `Generic`, the list of **Target function library** values shows all registered TFLs.

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: `TargetFunctionLibrary`
Type: string
Value: `'ANSI_C'` | `'C99 (ISO)'` | `'GNU99 (GNU)'` | `'C++ (ISO)'`
Default: `'ANSI_C'`

Recommended Settings

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

See Also

"Specifying Target Interfaces"

Shared code placement

Specify the location for generating utility functions, exported data type definitions, and declarations of exported data with custom storage class.

Settings

Default: `Auto`

`Auto`

Operates as follows:

- When the model contains Model blocks, places utility code within the `slprj/target/_sharedutils` folder.
- When the model does not contain Model blocks, places utility code in the build folder (generally, in *`model`*`.c` or *`model`*`.cpp`).

`Shared location`

Directs code for utilities to be placed within the `slprj` folder in your working folder.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	`Shared location`
Traceability	`Shared location`
Efficiency	No impact (execution, RAM) `Shared location` (ROM)
Safety precaution	No impact

See Also

- "Specifying Target Interfaces"
- "Shared Utility Code"

MAT-file logging

Specify whether to enable MAT-file logging.

Settings

Default: off

On

Enables MAT-file logging. When you select this option, the generated code saves to MAT-files simulation data specified in any of the following ways:

- **Configuration Parameters > Data Import/Export**, **Save to workspace** subpane (see "Data Import/Export Pane")
- To Workspace blocks
- Scope blocks with the **Save data to workspace** parameter enabled

In simulation, this data would be written to the MATLAB workspace, as described in "Exporting Simulation Data" and "Logging Signals with Scope Blocks". 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

Selecting this parameter enables **MAT-file variable name modifier**.

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

- "Logging"
- "Logging Data for Analysis"
- "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

This parameter is enabled by **MAT-file logging**.

Command-Line Information

Parameter: `LogVarNameModifier`
Type: string
Value: `'none'` | `'rt_'` | `'_rt'`
Default: `'rt_'`

Recommended Settings

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

See Also

- "Logging"
- "Logging Data for Analysis"

Code Format

Specify the code generation format.

Settings

Default: `RealTime`

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

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

Command-Line Information

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

Recommended Settings

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

See Also

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
-
-
-

Download to VxWorks target

Specify whether to automatically download the generated program to the VxWorks target.

Settings

Default: off

On

Automatically downloads the generated program to VxWorks after each build.

Off

Does not automatically download to VxWorks, you must downloaded generated programs manually.

Tips

- Automatic download requires specifying the target name and host name in the makefile, as described in .
- Before every build, reset VxWorks by pressing **Ctrl+X** on the host console or power-cycling the VxWorks chassis. This clears dangling processes or stale data that exists in VxWorks when the automatic download occurs.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	Off

See Also

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

Base task priority

Specify the priority with which the base rate task for the model is to be spawned.

Settings

Default: 30

Tips

- For a multirate, multitasking model, the Simulink Coder software increments the priority of each subrate task by one.
- The value you specify for this option will be overridden by a base priority specified in a call to the `rt_main()` function spawned as a task.

Command-Line Information

Parameter: `BasePriority`
Type: integer
Value: 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

•

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
-
-

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

"Host/Target Communication "

Transport layer

Specify the transport protocol for external mode communications.

Settings

Default: `tcpip`

`tcpip`
: Applies a TCP/IP transport mechanism. The MEX-file name is `ext_comm`.

Tip

The **MEX-file name** displayed next to **Transport layer** cannot be edited in the Configuration Parameters dialog box. For targets provided by MathWorks, the value is specified in *`matlabroot`*`/toolbox/simulink/simulink/extmode_transports.m`.

Dependency

This parameter is enabled by **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, `'myPuter'` or `'148.27.151.12'`)
- Verbosity level (`0` for no information or `1` for detailed information)
- TCP/IP server port number (an integer value between `256` and `65535`, with a default of `17725`)

Dependency

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

Command-Line Information

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

Recommended Settings

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

See Also

- "Target Interfacing"
- "Choosing Communications Protocol for Client and Server Interfaces"

Static memory allocation

Control the memory buffer for external mode communication.

Settings

Default: off

On

Enables the **Static memory buffer size** parameter for allocating allocate dynamic memory.

Off

Uses a static memory buffer for external mode instead of allocating dynamic memory (calls to malloc).

Tip

To determine how much memory you need to allocate, select verbose mode on the target to display the amount of memory it tries to allocate and the amount of memory available.

Dependencies

- This parameter is enabled by **External Mode**.
- This parameter enables **Static memory buffer size**.

Command-Line Information

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

Recommended Settings

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

See Also

"External Mode Interface Options"

Static memory buffer size

Specify the memory buffer size for external mode communication.

Settings

Default: `1000000`

Enter the number of bytes to preallocate for external mode communications buffers in the target.

Tips

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

Dependency

This parameter is enabled by **Static memory allocation**.

Command-Line Information

Parameter: `ExtModeStaticAllocSize`
Type: integer
Value: any valid value
Default: `1000000`

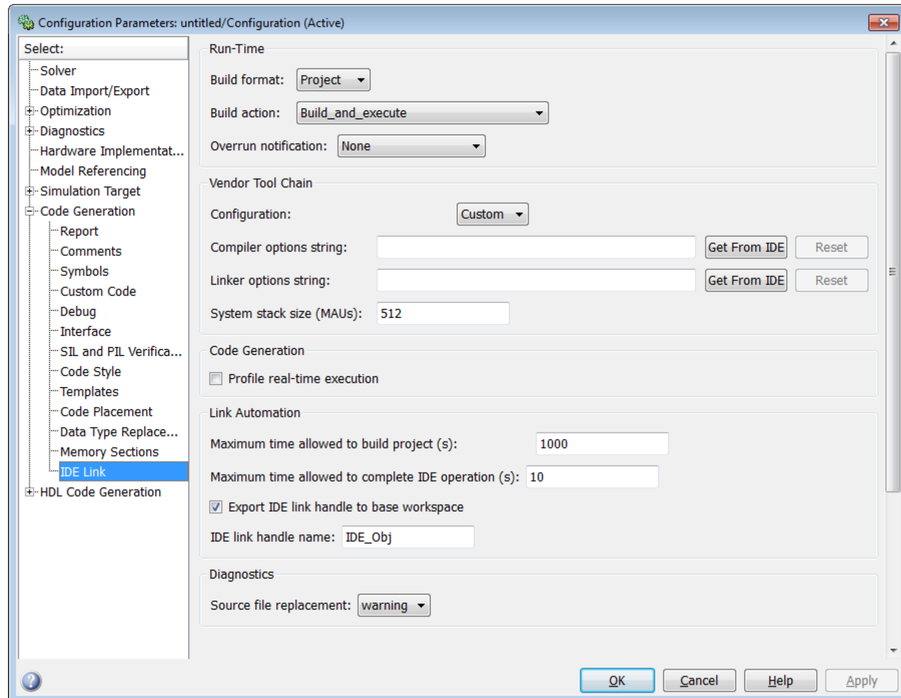
Recommended Settings

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

See Also

"External Mode Interface Options"

Code Generation Pane: IDE Link



In this section...

"Overview" on page 6-276
"Build format" on page 6-277
"Build action" on page 6-279
"Overrun notification" on page 6-282
"Function name" on page 6-284
"Configuration" on page 6-285
"Compiler options string" on page 6-287
"Linker options string" on page 6-289
"System stack size (MAUs)" on page 6-291

In this section...

"Profile real-time execution" on page 6-294

"Profile by" on page 6-296

"Number of profiling samples to collect" on page 6-298

"Maximum time allowed to build project (s)" on page 6-300

"Maximum time allowed to complete IDE operation (s)" on page 6-302

"Export IDE link handle to base workspace" on page 6-303

"IDE link handle name" on page 6-305

"Source file replacement" on page 6-306

Overview

Use this pane to configure the following parameters:

- Run-Time: set the build format to an IDE project or makefile, choose whether to build and execute the project, or create a PIL project.
- Vendor Tool Chain: set compiler and linker options.
- Code Generation: set options for profiling real-time execution.
- Link Automation: Set the maximum time to build projects and complete IDE operations. Set a default name for the IDE link handle.
- Diagnostics: Select the type of message to generate when the software replaces source files.

Build format

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: `Project`

`Project`
: Builds your model as an IDE project.

`Makefile`
: Creates a makefile and uses it to build your model.

Dependencies

Selecting `Makefile` removes the following parameters:

- **Code Generation**
  - **Profile real-time execution**
  - **Profile by**
  - **Number of profiling samples to collect**
- **Link Automation**
  - **Maximum time allowed to build project (s)**
  - **Maximum time allowed to complete IDE operation (s)**
  - **Export IDE link handle to base workspace**
  - **IDE link handle name**

Command-Line Information

Parameter: `buildFormat`
Type: string
Value: `Project | Makefile`
Default: `Build_and_execute`

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: `Build_and_execute`

If you set **Build format** to `Project`, select one of the following options:

`Build_and_execute`
: Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

`Create_project`
: Directs Simulink Coder software to create a new project in the IDE. The command line equivalent for this setting is `Create`.

`Archive_library`
: Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

`Build`
: Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

`Create_processor_in_the_loop_project`
: Directs the Simulink Coder code generation process to create PIL algorithm object code as part of the project build.

If you set **Build format** to `Makefile`, select one of the following options:

`Create_makefile`
: Creates a makefile. For example, ".mk". The command line equivalent for this setting is `Create`.

`Archive_library`
: Creates a makefile and an archive library. For example, ".a" or ".lib".

`Build`
: Creates a makefile and an executable. For example, ".exe".

`Build_and_execute`
    Creates a makefile and an executable. Then it evaluates the execute instruction under the **Execute** tab in the current XMakefile configuration.

Dependencies

Selecting `Archive_library` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace**

Selecting `Create_processor_in_the_loop_project` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

Command-Line Information

Parameter: `buildAction`
Type: string
Value: `Build` | `Build_and_execute` | `Create` | `Archive_library` | `Create_processor_in_the_loop_project`
Default: `Build_and_execute`

Recommended Settings

Application	Setting
Debugging	`Build_and_execute`
Traceability	`Archive_library`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

For more information about PIL and its uses, refer to the "Verifying Generated Code via Processor-in-the-Loop" topic.

Overrun notification

Specifies how your program responds to overrun conditions during execution.

Settings

Default: `None`

`None`
: Your program does not notify you when it encounters an overrun condition.

`Print_message`
: Your program prints a message to standard output when it encounters an overrun condition.

`Call_custom_function`
: When your program encounters an overrun condition, it executes a function that you specify in **Function name**.

Tips

- The definition of the standard output depends on your configuration.

Dependencies

Selecting `Call_custom_function` enables the **Function name** parameter.

Setting this parameter to `Call_custom_function` enables the **Function name** parameter.

Command-Line Information

Parameter: `overrunNotificationMethod`
Type: string
Value: `None` | `Print_message` | `Call_custom_function`
Default: `None`

Recommended Settings

Application	Setting
Debugging	`Print_message` or `Call_custom_function`
Traceability	`Print_message`
Efficiency	`None`
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Function name

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

Settings

No Default

Dependencies

This parameter is enabled by setting **Overrun notification** to `Call_custom_function`.

Command-Line Information

Parameter: `overrunNotificationFcn`
Type: string
Value: no default
Default: no default

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Configuration

Sets the Configuration for building your project from the model.

Settings

Default: `Custom`

`Custom`
: Lets the user apply a specialized combination of build and optimization settings.

  `Custom` applies the same settings as the Release project configuration in IDE, except:

  - The compiler options do not use any optimizations.
  - The memory configuration specifies a memory model that uses `Far Aggregate` for data and `Far` for functions.

`Debug`
: Applies the `Debug` Configuration defined by the IDE to the generated project and code.

`Release`
: Applies the `Release` project configuration defined by the IDE to the generated project and code.

Dependencies

- Selecting `Custom` disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting `Release` sets the **Compiler options string** to the settings defined by the IDE.
- Selecting `Debug` sets the **Compiler options string** to the settings defined by the IDE.

.

Command-Line Information

Parameter: `projectOptions`

Type: string
Value: `Custom` | `Debug` | `Release`
Default: `Custom`

Recommended Settings

Application	Setting
Debugging	`Custom` or `Debug`
Traceability	`Custom`, `Debug`, `Release`
Efficiency	`Release`
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the coder product does not set any optimization flags.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Configuration** to `Custom` applies the `Custom` compiler options defined by coder software. `Custom` does not use any optimizations.
- Setting **Configuration** to `Debug` applies the debug settings defined by the IDE.
- Setting **Configuration** to `Release` applies the release settings defined by the IDE.

Command-Line Information

Parameter: `compilerOptionsStr`
Type: string
Value: `Custom` | `Debug` | `Release`
Default: `Custom`

Recommended Settings

Application	Setting
Debugging	`Custom`
Traceability	`Custom`

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Linker options string

To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, the coder product does not set any linker options.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information

Parameter: `linkerOptionsStr`
Type: string
Value: any valid linker option
Default: none

Recommended Settings

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

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory.

This parameter is used in all targets to allocate the stack size for the generated application. For example, with embedded processors that are not running an operating system, this parameter determines the total stack space that can be used for the application. For operating systems such as Linux or WindowsVxWorks, this value specifies the stack space allocated per thread.

This parameter also affects the "Maximum stack size (bytes)" parameter, located in the Optimization > Signals and Parameters pane.

Settings

Default: 8192

Minimum: 0

Maximum: Available memory

- Enter the stack size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.
- The software does not verify the value you entered is valid. Enter the correct value.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

When you set the **System target file** parameter on the **Code Generation** pane to `idelink_ert.tlc` or `idelink_grt.tlc`, the software sets the **Maximum stack size** parameter on the **Optimization > Signals and Parameters** pane to `Inherit from target` and makes it non-editable. In that case, the **Maximum stack size** parameter compares the value of (**System stack size**/2) with 200,000 bytes and uses the smaller of the two values.

Command-Line Information

Parameter: `systemStackSize`
Type: int
Default: `8192`

Recommended Settings

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

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Profile real-time execution

Enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

Settings

Default: Off

On

Adds instrumentation to the generated code to support execution profiling and generate the profiling report.

Off

Does not instrument the generated code to produce the profile report.

Dependencies

This parameter adds **Number of profiling samples to collect** and **Profile by**.

Selecting this parameter enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop project` removes this parameter.

Command-Line Information

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

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics..

Profile by

Defines which execution profiling technique to use.

Settings

Default: `Task`

`Task`
  Profiles model execution by the tasks in the model.

`Atomic subsystem`
  Profiles model execution by the atomic subsystems in the model.

Dependencies

Selecting **Real-time execution profiling** enables this parameter.

Command-Line Information

Parameter: `profileBy`
Type: string
Value: `Task` | `Atomic subsystem`
Default: `Task`

Recommended Settings

Application	Setting
Debugging	`Task` or `Atomic subsystem`
Traceability	`Archive_library`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

For more information about PIL and its uses, refer to the "Verifying Generated Code via Processor-in-the-Loop" topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics.

Number of profiling samples to collect

Specify the size of the buffer that holds the profiling samples. Enter a value that is 2 times the number of profiling samples.

Each task or subsystem execution instance represents one profiling sample. Each sample requires two memory locations, one for the start time and one for the end time. Consequently, the size of the buffer is twice the number of samples.

Sample collection begins with the start of code execution and ends when the buffer is full.

The profiling data is held in a statically sited buffer on the target processor.

Settings

Default: 100

Minimum: 2

Maximum: Buffer capacity

Tips

- Data collection stops when the buffer is full, but the application and processor continue running.
- Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

Dependencies

This parameter is enabled by **Profile real-time execution**.

Command-Line Information

Parameter:`ProfileNumSamples`
Type: int
Value: Positive integer
Default: `100`

Recommended Settings

Application	Setting
Debugging	100
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

Settings

Default: 1000

Minimum: 1

Maximum: No limit

Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to complete IDE operation** timeout value.

Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

Command-Line Information

Parameter:`ideObjBuildTimeout`
Type: int
Value: Integer greater than 0
Default: `100`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Maximum time allowed to complete IDE operation (s)

specifies how long, in seconds, the software waits for IDE functions, such as `read` or `write`, to return completion messages.

Settings

Default: 10

Minimum: 1

Maximum: No limit

Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to build project (s)** timeout value

Command-Line Information

Parameter: `'ideObjTimeout'`
Type: int
Value:
Default: `10`

Recommended Settings

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

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Export IDE link handle to base workspace

Directs the software to export the IDE_Obj object to your MATLAB workspace.

Settings

Default: On

On

Directs the build process to export the IDE_Obj object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.

Off

prevents the build process from exporting the IDE_Obj object to your MATLAB software workspace.

Dependency

Selecting **Profile real-time execution** enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Selecting **Export IDE link handle to base workspace** enables **IDE link handle name**.

Command-Line Information

Parameter: exportIDEObj
Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

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

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

IDE link handle name

specifies the name of the IDE_Obj object that the build process creates.

Settings

Default: IDE_Obj

- Enter any valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the IDE_Obj object.
- The handle name is case sensitive.

Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

Command-Line Information

Parameter: ideObjName
Type: string
Value:
Default: IDE_Obj

Recommended Settings

Application	Setting
Debugging	Enter any valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Source file replacement

Selects the diagnostic action to take if the coder software detects conflicts that you are replacing source code with custom code.

Settings

Default: `warn`

`none`
: Does not generate warnings or errors when it finds conflicts.

`warning`
: Displays a warning.

`error`
: Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

Tips

- The build operation continues if you select `warning` and the software detects custom code replacement. You see warning messages as the build progresses.
- Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select `none` when the replacement process is correct and you do not want to see multiple messages during your build.
- The messages apply to Simulink Coder **Custom Code** replacement options as well.

Command-Line Information

Parameter: `DiagnosticActions`
Type: string
Value: `none | warning | error`
Default: `warning`

Recommended Settings

Application	Setting
Debugging	`error`
Traceability	`error`
Efficiency	`warning`
Safety precaution	`error`

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Parameter Reference

In this section...
"Recommended Settings Summary" on page 6-308
"Parameter Command-Line Information Summary" on page 6-337

Recommended Settings Summary

The following table summarizes the impact of each configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the GRT and ERT targets, unless otherwise specified.

For parameters that are available only when an ERT target is specified, see the "Recommended Settings Summary" in the Embedded Coder documentation.

For additional details, click the links in the Configuration Parameter column.

Mapping Application Requirements to the Solver Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Start Time	No impact	No impact	No impact	0.0	0.0 seconds
Stop time	No impact	No impact	No impact	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
"Record and inspect simulation output"	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: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Block reduction	Off (GRT) No impact (ERT)	Off	On	Off	On
Implement logic signals as Boolean data (vs. double)	No impact	No impact	On	On	On
Conditional input branch execution	No impact	On	On (execution) No impact (ROM, RAM)	Off	On
Application lifespan (days)	No impact	No impact	Finite value	`inf`	`inf`
Use memset to initialize floats and doubles to 0.0	No impact	No impact	On* (execution, ROM) No impact (RAM)	No impact	On
Use floating-point multiplication to handle net slope corrections	No impact	No impact	On (when target hardware supports efficient multiplication) Off (otherwise)	Off	Off

Mapping Application Requirements to the Optimization Pane: General Tab (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 (execution, ROM) No impact (RAM)	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
*The command-line value is reverse of the listed value.					

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Inline parameters	Off (GRT) On (ERT)	On	On	No impact	Off
Signal storage reuse	Off	Off	On	No impact	On

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Enable local block outputs	Off	No impact	On	No impact	On
Eliminate superfluous local variables (Expression folding)	Off	No impact (GRT) Off (ERT)	On	No impact	On
"Minimize data copies between local and global variables"	Off	Off	No impact (execution) On (ROM, RAM)	No impact	Off
Loop unrolling threshold	No impact	No impact	>0	>1	5
Maximum stack size (bytes)	No impact	No impact	No impact	No impact	`Inherit from target`
Use memcpy for vector assignment	No impact	No impact	On	No impact	On
Memcpy threshold (bytes)	No impact	No impact	Accept default or determine target-specific optimal value	No impact	64

Mapping Application Requirements to the Optimization Pane: Signals and Parameters Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
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: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Use bitsets for storing state configuration"	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off
"Use bitsets for storing Boolean data"	Off	Off	Off (execution, ROM) On (RAM)	No impact	Off

Mapping Application Requirements to the Diagnostics Pane: Solver Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Algebraic loop"	`error`	No impact	No impact	`error`	`warning`
"Minimize algebraic loop"	No impact	No impact	No impact	`error`	`warning`
"Block priority violation"	No impact	No impact	No impact	`error`	`warning`
"Consecutive zero-crossings violation"	No impact	No impact	No impact	`warning` or `error`	`error`
"Unspecified inheritability of sample time"	No impact	No impact	No impact	`error`	`warning`
"Solver data inconsistency"	`warning`	No impact	`none`	No impact	`warning`
"Automatic solver parameter selection"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Sample Time Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Source block specifies -1 sample time"	No impact	No impact	No impact	`error`	`none`
"Discrete used as continuous"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Sample Time Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Multitask rate transition"	No impact	No impact	No impact	`error`	`error`
"Single task rate transition"	No impact	No impact	No impact	`none` or `error`	`none`
"Multitask conditionally executed subsystem"	No impact	No impact	No impact	`error`	`error`
"Tasks with equal priority"	No impact	No impact	No impact	`none` or `error`	`warning`
"Enforce sample times specified by Signal Specification blocks"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Signal resolution"	No impact	No impact	No impact	`Explicit only`	`Explicit only`
"Division by singular matrix"	No impact	No impact	No impact	`error`	`none`
"Underspecified data types"	No impact	No impact	No impact	`error`	`none`

Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Simulation range checking"	warning or error	warning or error	none	error	none
"Detect overflow"	No impact	No impact	No impact	error	warning
"Inf or NaN block output"	No impact	No impact	No impact	error	none
""rt" prefix for identifiers"	No impact	No impact	No impact	error	error
"Detect downcast"	No impact	No impact	No impact	error	error
"Detect overflow"	No impact	No impact	No impact	error	error
"Detect underflow"	No impact	No impact	No impact	error	none
"Detect precision loss"	No impact	No impact	No impact	error	error
"Detect loss of tunability"	No impact	No impact	No impact	error	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

Mapping Application Requirements to the Diagnostics Pane: Data Validity Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Duplicate data store names"	`warning`	No impact	`none`	No impact	`none`
"Check undefined subsystem initial output"	No impact	No impact	No impact	On	On
"Check preactivation output of execution context"	No impact	No impact	No impact	On	Off
"Check runtime output of 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 Pane: Type Conversion Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Unnecessary type conversions"	No impact	No impact	No impact	warning	none
"Vector/matrix block input conversion"	No impact	No impact	No impact	error	none
"32-bit integer to single precision float conversion"	No impact	No impact	No impact	warning	warning

Mapping Application Requirements to the Diagnostics Pane: Connectivity Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Signal label mismatch"	No impact	No impact	No impact	error	none
"Unconnected block input ports"	No impact	No impact	No impact	error	warning
"Unconnected block output ports"	No impact	No impact	No impact	error	warning
"Unconnected line"	No impact	No impact	No impact	error	none
"Unspecified bus object at root Outport block"	No impact	No impact	No impact	error	warning

Mapping Application Requirements to the Diagnostics Pane: Connectivity Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"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 Pane: Compatibility Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"S-function upgrades needed"	No impact	No impact	No impact	`error`	`none`

Mapping Application Requirements to the Diagnostics Pane: Model Referencing Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Model block version mismatch"	No impact	No impact	No impact	`none`	`none`
"Port and parameter mismatch"	No impact	No impact	No impact	`error`	`none`
"Model configuration mismatch"	No impact	No impact	No impact	`warning`	`none`
"Invalid root Inport/Outport block connection"	No impact	No impact	No impact	`error`	`none`
"Unsupported data logging"	No impact	No impact	No impact	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Saving Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Block diagram contains disabled library links"	No impact	No impact	No impact	No impact	`warning`
"Block diagram contains parameterized library links"	No impact	No impact	No impact	No impact	`none`

Mapping Application Requirements to the Diagnostics Pane: Stateflow Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Unused data and events"	`warning`	No impact	No impact (for simulation and during development) `none` (for production code generation)	`warning`	`warning`
"Unexpected backtracking"	`warning`	No impact	No impact	`error`	`warning`
"Invalid input data access in chart initialization"	`warning`	No impact	No impact	`error`	`warning`
"No unconditional default transitions"	`warning`	No impact	No impact (for simulation and during development) `none` (for production	`error`	`warning`

Mapping Application Requirements to the Diagnostics Pane: Stateflow Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
			code generation)		
"Transition outside natural parent"	`warning`	No impact	No impact (for simulation and during development) `none` (for production code generation)	`error`	`warning`

Mapping Application Requirements to the Hardware Implementation Pane

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Device vendor	No impact	No impact	No impact	No impact	`Generic`
Device type	No impact	No impact	No impact	No impact	`Unspecified (assume 32 bit Generic)`

Mapping Application Requirements to the Hardware Implementation Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Number of bits	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for code generation	`char` 8, `short` 16, `int` 32, `long` 32, `native` 32
Largest atomic size	No impact	No impact	Target specific	No impact for simulation and during development Match operation of compiler and hardware for code generation	integer `Char`, floating-point `None`
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 for simulation and during development `Undefined` for production code generation	No impact for simulation and during development `Zero` or `Floor` for production code generation	No impact for simulation and during development `Zero` for production code generation	No impact for simulation and during development `Floor` for production code generation	`Undefined`
Shift right on a signed integer as arithmetic shift	No impact	No impact	On	No impact	On
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”	No impact	No impact	No impact	`If any changes detected` or `Never` If you use the `Never` setting,	`If any changes detected`

Mapping Application Requirements to the Model Referencing Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
				then set the **Never rebuild diagnostic** parameter to `Error if rebuild required`	
“Never rebuild diagnostic”	No impact	No impact	No impact	`error if rebuild required`	`error if rebuild required`
“Enable parallel model reference builds”	No impact	No impact	No impact	No impact	Off
“MATLAB worker initialization for builds”	No impact	No impact	No impact	No impact	`None`
“Total number of instances allowed per top model”	No impact	No impact	No impact	No impact	`Multiple`
“Pass fixed-size scalar root inputs by value for code generation”	No impact	No impact	No impact	Off	Off
“Minimize algebraic loop occurrences”	No impact	No impact	No impact	Off	Off

Mapping Application Requirements to the Model Referencing Pane (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Propagate sizes of variable-size signals"	No impact	No impact	No impact	Off	`Infer from blocks in model`
"Model dependencies"	No impact	No impact	No impact	No impact	`''`

Mapping Application Requirements to the Simulation Target Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Enable debugging/animation"	On	No impact	Off	On	On
"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 Pane: General Tab (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 Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Reserved names"	No impact	No impact	No impact	No impact	`{}`

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Parse custom code symbols"	On	No impact	No impact	On	On
"Source file"	No impact	No impact	No impact	No impact	`''`
"Header file"	No impact	No impact	No impact	No impact	`''`
"Initialize function"	No impact	No impact	No impact	No impact	`''`

Mapping Application Requirements to the Simulation Target Pane: Custom Code Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
"Terminate function"	No impact	No impact	No impact	No impact	`''`
"Include directories"	No impact	No impact	No impact	No impact	`''`
"Source files"	No impact	No impact	No impact	No impact	`''`
"Libraries"	No impact	No impact	No impact	No impact	`''`

Mapping Application Requirements to the Code Generation Pane: General Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
System target file	No impact	No impact	No impact	No impact (GRT) ERT based (ERT)	`grt.tlc`
Language	No impact	No impact	No impact	No impact	C
Compiler optimization level	`Optimizations off (faster builds)`	`Optimizations off (faster builds)`	`Optimizations on (faster runs)` (execution) No impact (ROM, RAM)	No impact	`Optimizations off (faster builds)`

Mapping Application Requirements to the Code Generation Pane: General Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
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	`''`
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	`Execution efficiency`	Not applicable for GRT-based targets	`Unspecified`
"Check model before generating code" on page 6-33	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`On (proceed with warnings)` or `On (stop for warnings)`	`Off`
Generate code only	Off	No impact	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Report Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precautions	
"Create code generation report" on page 6-41	On	On	No impact	On	Off
"Launch report automatically" on page 6-44	On	On	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Comments Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Include comments	On	On	No impact	On	On
Simulink block / Stateflow object comments	On	On	No impact	On	On
Show eliminated blocks	On	On	No impact	On	Off
Verbose comments for Simulink Global storage class	On	On	No impact	On	Off

Mapping Application Requirements to the Code Generation Pane: Symbols Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Maximum identifier length	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 Code Generation Pane: Custom Code Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Use the same custom code settings as Simulation Target	No impact	No impact	No impact	No impact	Off
Source file	No impact	No impact	No impact	No impact	''
Header file	No impact	No impact	No impact	No impact	''
Initialize function	No impact	No impact	No impact	No impact	''
Terminate function	No impact	No impact	No impact	No impact	''
Include directories	No impact	No impact	No impact	No impact	''

Mapping Application Requirements to the Code Generation Pane: Custom Code Tab (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 Code Generation Pane: Debug Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Verbose build	On	No impact	No impact	On	On
Retain .rtw file	On	No impact	No impact	No impact	Off
"Profile TLC" on page 6-145	On	No impact	No impact	No impact	Off
Start TLC debugger when generating code	On	No impact	No impact	No impact	Off
Start TLC coverage when generating code	On	No impact	No impact	No impact	Off
Enable TLC assertion	On	No impact	No impact	On	Off

Mapping Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
Target function library	No impact	No impact	Any valid value	No impact	`C89/C90 (ANSI)`
Shared code placement	`Shared location` (GRT) No impact (ERT)	`Shared location` (GRT) No impact (ERT)	No impact (execution, RAM) `Shared location` (ROM)	No impact	`Auto`
Support non-finite numbers	No impact	No impact	Off (Execution, ROM) No impact (RAM)	Off	On
MAT-file logging	On	No impact	Off	Off	On (GRT) Off (ERT)
MAT-file variable name modifier	No impact	No impact	No impact	No impact	`rt_`
Interface	No impact	No impact	No impact	No impact (GRT) `None` (ERT)	`None`
Generate C API for: signals	No impact	No impact	No impact	No impact	On
Generate C API for: parameters	No impact	No impact	No impact	No impact	On
Generate C API for: states	No impact	No impact	No impact	No impact	Off

Mapping Application Requirements to the Code Generation Pane: Interface Tab (Continued)

Configuration Parameter	Settings for Building Code				Factory Default
	Debugging	Traceability	Efficiency	Safety Precaution	
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 6-272	No impact	No impact	No impact	No impact	1000000

Parameter Command-Line Information Summary

The following table lists Simulink Coder parameters that you can use to tune model and target configurations. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to Configuration Parameter dialog box equivalents. For descriptions of the panes and options in that dialog box, see Configuration Parameters in the Simulink Coder documentation.

Use the `get_param` and `set_param` commands to retrieve and set the values of the parameters on the MATLAB command line or programmatically in scripts.

The Configuration Wizard in the Embedded Coder product provides buttons and scripts for customizing code generation. See "Using Configuration Wizard Blocks" in the 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 "Tuning Parameters".

For parameters that are specific to the ERT target, or targets based on the ERT target, see "Parameter Command-Line Information Summary" in the Embedded Coder documentation.

Note Parameters that are specific to Stateflow or 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: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`BooleanDataType` `off`, **`on`**	**Optimization > Implement logic signals as Boolean data (vs. double)**	Control the output data type of blocks that generate logic signals.
`EfficientFloat2IntCast` **`off`**, `on`	**Optimization > Remove code from floating-point to integer conversions that wrap out-of-range values**	Remove wrapping code that handles out-of-range floating-point to integer conversion results.
`EfficientMapNaN2IntZero` `off`, **`on`**	**Optimization > Remove code from floating-point to integer conversions with saturation that maps NaN to zero**	Remove code that handles floating-point to integer conversion results for `NaN` values.
`InitFltsAndDblsToZero` **`off`**, `on`	**Optimization > Use memset to initialize floats and doubles to 0.0**	Optimize initialization of storage for `float` and `double` values. Set this option if the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern 0.
`LifeSpan` *`string`*	**Optimization > Application lifespan (days)**	Optimize the size of counters used to compute absolute and elapsed time, using the specified application life span value.
`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.

Command-Line Information: Optimization Pane: General Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`UseFloatMulNetSlope` (Simulink Fixed Point) **`off`**, `on`	**Optimization > Use floating-point multiplication to handle net slope corrections**	Use floating-point multiplication to perform net slope correction for floating-point to fixed-point casts.
`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: Optimization Pane: Signals and Parameters Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`BufferReuse` `off`, **`on`**	**Optimization > Signals and Parameters > Reuse block outputs**	Reuse local (function) variables for block outputs wherever possible. Selecting this option trades code traceability for code efficiency.
`EnableMemcpy` `off`, **`on`**	**Optimization > Signals and Parameters > Use memcpy for vector assignment**	Optimize code generated for vector assignment by replacing `for` loops with `memcpy` function calls.

Command-Line Information: Optimization Pane: Signals and Parameters Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
EnhancedBackFolding **off**, on	**Optimization > Signals and Parameters > Minimize data copies between local and global variables**	Reuse existing global variables to store temporary results.
ExpressionFolding off, **on**	**Optimization > Signals and Parameters > Eliminate superfluous local variables (Expression folding) > Interface**	Collapse block computations into single expressions wherever possible. This improves code readability and efficiency.
InlineInvariantSignals off, **on**	**Optimization > Signals and Parameters > Inline invariant signals**	Precompute and inline the values of invariant signals in the generated code.
LocalBlockOutputs off, **on**	**Optimization > Signals and Parameters > Enable local block outputs**	Declare block outputs in local (function) scope wherever possible to reduce global RAM usage.
MemcpyThreshold int - **64**	**Optimization > Signals and Parameters > Memcpy threshold (bytes)**	Specify the minimum array size in bytes for which memcpy function calls should replace for loops in the generated code for vector assignments.

Command-Line Information: Optimization Pane: Signals and Parameters Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`RollThreshold` `int` - **`5`**	**Optimization > Signals and Parameters > Loop unrolling threshold**	Specify the minimum signal width for which a `for` loop is to be generated.
`MaxStackSize` `<Specify a value>`, **`Inherit from target`**	**Optimization > Signals and Parameters > Maximum stack size (bytes)**	Specify the maximum stack size in bytes for your model.

Command-Line Information: Optimization Pane: Stateflow Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`DataBitsets` (Stateflow) **`off`**, `on`	**Optimization > Stateflow > Use bitsets for storing Boolean data**	Use bit sets for storing Boolean data.
`StateBitsets` (Stateflow) **`off`**, `on`	**Optimization > Stateflow > Use bitsets for storing state configuration**	Use bit sets for storing state configuration.

Command-Line Information: Code Generation Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CheckMdlBeforeBuild` *string* - **`off`**, `warning`, `error`	**Code Generation > Check model before generating code**	Specify whether to run Code Generation Advisor checks before generating code.
`GenCodeOnly` *string* - **`off`**, `on`	**Code Generation > Generate code only**	Generate source code, but do not execute the makefile to build an executable.
`GenerateMakefile` *string* - `off`, **`on`**	**Code Generation > Generate makefile**	Specify whether to generate a makefile during the build process for a model.
`MakeCommand` *string* - **`make_rtw`**	**Code Generation > Make command**	Specify the `make` command and optional arguments to be used to generate an executable for the model.
`ObjectivePriorities` (GRT) string - **`{''}`**, `{'Debugging'}`, `{'Execution efficiency'}`	**Code Generation > Select objective**	Specify the code generation objectives to use with the Code Generation Advisor.
`ObjectivePriorities` (ERT) string - **`{''}`**, `{'Efficiency'}`, `{'Traceability'}`, `{'Safety precaution'}`, `{'Debugging'}`	**Code Generation > Set objectives**	Specify and prioritize the code generation objectives to use with the Code Generation Advisor.

Command-Line Information: Code Generation Pane: General Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`RTWCompilerOptimization` string - **`Off`**, `On`, `Custom`	**Code Generation > Compiler optimization level**	Use this parameter to trade off compilation time against run time for your model code without having to supply compiler-specific flags to other levels of the Simulink Coder build process. `Off` - Turn compiler optimizations off for faster builds `On` - Turn compiler optimizations on for faster code execution `Custom` - Specify custom compiler optimization flags via the `RTWCustomCompilerOptimizations` parameter
`RTWCustomCompiler Optimizations` `string` - `''`, unquoted string of compiler optimization flags	**Code Generation > Custom compiler optimization flags**	If you specified `Custom` to the `RTWCompilerOptimization` parameter, use this parameter to specify custom compiler optimization flags, for example, `-O2`.
`SaveLog` **`off`**, `on`	**Code Generation > Save build log**	Save build log.
`SystemTargetFile` `string` - **`grt.tlc`**	**Code Generation > System target file**	Specify a system target file.

Command-Line Information: Code Generation Pane: General Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
TargetLang string - **C**, C++, C++ (Encapsulated) (ERT)	**Code Generation > 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 an Embedded Coder license.
TemplateMakefile *string* - **grt_default_tmf**	**Code Generation > Template makefile**	Specify the current template makefile for building a Simulink Coder target.
TLCOptions *string* - ''	**Code Generation > TLC options**	Specify additional TLC command line options.

Command-Line Information: Code Generation Pane: Report Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
GenerateReport string - **off**, on	**Code Generation > Report > Create code generation report**	Document the generated C or C++ code in an HTML report.
LaunchReport string - **off**, on	**Code Generation > Report > Launch report automatically**	Display the HTML report after code generation completes.

Command-Line Information: Code Generation Pane: Comments Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ForceParamTrailComments` *string* - **`off`**, `on`	**Code Generation > Comments > Verbose comments for SimulinkGlobal storage class**	Specify that comments be included in the generated file. To reduce file size, the model parameters data structure is not commented when there are more than 1000 parameters.
`GenerateComments` *string* - `off`, **`on`**	**Code Generation > Comments > Include comments**	Include comments in generated code.
`ShowEliminatedStatement` *string* - **`off`**, `on`	**Code Generation > Comments > Show eliminated blocks**	Show statements for eliminated blocks as comments in the generated code.
`SimulinkBlockComments` *string* - `off`, **`on`**	**Code Generation > Comments > Simulink block / Stateflow object comments**	Insert Simulink block and Stateflow object names as comments above the generated code for each block.

Command-Line Information: Code Generation Pane: Symbols Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MaxIdLength` `int - **31**`	**Code Generation > Symbols > Maximum identifier length**	Specify the maximum number of characters that can be used in generated function, type definition, and variable names.
`ReservedNameArray` *string array* - **{}**	**Code Generation > Symbols > Reserved names**	Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code to avoid name conflicts.
`UseSimReservedNames` `string` - **`off`**, `on`	**Code Generation > 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: Code Generation Pane: Custom Code Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomHeaderCode` *string* - `''`	**Code Generation > Custom Code > Header file**	Specify code to appear near the top of the generated model header file.
`CustomInclude` *string* - `''`	**Code Generation > Custom Code > Include directories**	Specify a space-separated list of include folders to add to the include path when compiling the generated code.

Command-Line Information: Code Generation Pane: Custom Code Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		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, `'C:\Project "C:\Custom Files"'`
`CustomInitializer` *string* - `''`	**Code Generation > Custom Code**	Specify code to appear in the generated model initialize function.
`CustomLibrary` *string* - `''`	**Code Generation > Custom Code > Initialize function Libraries**	Specify a space-separated list of static library files to link with the generated code.
`CustomSource` *string* - `''`	**Code Generation > Custom Code > Source files**	Specify a space-separated list of source files to compile and link with the generated code.
`CustomSourceCode` *string* - `''`	**Code Generation > Custom Code > Source file**	Specify code to appear near the top of the generated model source file.

Command-Line Information: Code Generation Pane: Custom Code Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CustomTerminator *string* - ''	**Code Generation > Custom Code > Terminate function**	Specify code to appear in the generated model terminate function.
RTWUseSimCustomCode string - **off**, on	**Code Generation > Custom Code > Use the same custom code settings as Simulation Target**	Specify whether to use the same custom code settings as those in the **Simulation Target > Custom Code** pane.

Command-Line Information: Code Generation Pane: Debug Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ProfileTLC string - **off**, on	**Code Generation > Debug > Profile TLC**	Profile the execution time of each TLC file used to generate code for this model in HTML format.
RTWVerbose string - off, **on**	**Code Generation > Debug > Verbose build**	Display messages indicating code generation stages and compiler output.
RetainRTWFile string - **off**, on	**Code Generation > Debug > Retain .rtw file**	Retain the *model*.rtw file in the current build folder.
TLCAssert string - **off**, on	**Code Generation > Debug > Enable TLC assertion**	Produce a TLC stack trace when the argument to the assert directives evaluates to false.

Command-Line Information: Code Generation Pane: Debug Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TLCCoverage` string - **`off`**, `on`	**Code Generation > Debug > Start TLC coverage when generating code**	Generate `.log` files containing the number of times each line of TLC code is executed during code generation.
`TLCDebug` string - **`off`**, `on`	**Code Generation > Debug > Start TLC debugger when generating code**	Start the TLC debugger during code generation at the beginning of the TLC program. TLC breakpoint statements automatically invoke the TLC debugger regardless of this setting.

Command-Line Information: Code Generation Pane: Interface Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ExtMode` **`off`**, `on`	**Code Generation > Interface > Interface**	Specify the data interface to be generated with the code.
`ExtModeMexArgs` *`string`* `('')`	**Code Generation > 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`	**Code Generation > Interface > Static memory allocation**	Use a static memory buffer for external mode instead of allocating dynamic memory (calls to `malloc`).

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ExtModeStaticAllocSize *integer* (**1000000**)	**Code Generation > Interface > Static memory buffer size**	Specify the size in bytes of the external mode static memory buffer.
ExtModeTransport int - **0** for TCP/IP, 1 for 32-bit Windows serial	**Code Generation > Interface > Interface > External > Transport layer**	Specify transport protocols for external mode communications.
GenerateASAP2 **off**, on	**Code Generation > Interface > Interface**	Specify the data interface to be generated with the code.
TargetFunctionLibrary string - **ANSI_C**, C99 (ISO), GNU99 (GNU), C++ (ISO) (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.)	**Code Generation > Interface > Target function library**	Specify a target-specific math library for your model. Verify that your compiler supports the library you want to use; otherwise compile-time errors can occur. ANSI_C - ISO/IEC 9899:1990 C standard math library for floating-point functions C99 (ISO) - ISO/IEC 9899:1999 C standard math library GNU99 (GNU) - GNU gcc math library, which provides C99 extensions as defined by compiler option -std=gnu99 C++ (ISO) - ISO/IEC 14882:2003 C++ standard math library

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
LogVarNameModifier string - **none**, rt_, _rt	**Code Generation > Interface > MAT-file variable name modifier**	Augment the MAT-file variable name.
MatFileLogging string - off, on (Default is on for GRT targets, off for ERT targets)	**Code Generation > Interface > MAT-file logging**	Generate code that logs data to a MAT-file.
RTWCAPIParams string - **off**, on	**Code Generation > Interface > Generate C API for: parameters**	Generate C API parameter tuning structures.
RTWCAPISignals string - **off**, on	**Code Generation > Interface > Generate C API for: signals**	Generate C API signal structure.
RTWCAPIStates string - **off**, on	**Code Generation > Interface > Generate C API for: states**	Generate C API state structure.
SupportNonFinite string - off, **on**	**Code Generation > Interface > Support non-finite numbers**	Support nonfinite values (inf, nan, -inf) in the generated code.
UtilityFuncGeneration string - **Auto**, Shared location	**Code Generation > Interface > Shared code placement**	Specify where utility code is to be generated.

Command-Line Information: Not in GUI

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CodeGenDirectory`	Not available	For MathWorks use only.
`Comment`	Not available	For MathWorks use only.
`CompOptLevelCompliant` `off`, `on`	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to use the **Compiler optimization level** parameter on the **Code Generation** pane to control the compiler optimization level for building generated code. Default is `off` for custom targets and `on` for targets provided with the Simulink Coder and Embedded Coder products.
`ConfigAtBuild`	Not available	For MathWorks use only.
`ConfigurationMode`	Not available	For MathWorks use only.
`ConfigurationScript`	Not available	For MathWorks use only.
`ERTCustomFileBanners`	Not available	For MathWorks use only.
`EvaledLifeSpan`	Not available	For MathWorks use only.
`ExtModeMexFile`	Not available	For MathWorks use only.
`ExtModeTesting`	Not available	For MathWorks use only.
`FoldNonRolledExpr`	Not available	For MathWorks use only.
`GenerateFullHeader`	Not available	For MathWorks use only.
`IncAutoGenComments`	Not available	For MathWorks use only.
`IncludeRegionsInRTWFile BlockHierarchyMap`	Not available	For MathWorks use only.
`IncludeRootSignalInRTWFile`	Not available	For MathWorks use only.

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IncludeVirtualBlocksInRTWFileBlockHierarchyMap`	Not available	For MathWorks use only.
`IsERTTarget`	Not available	For MathWorks use only.
`IsPILTarget`	Not available	For MathWorks use only.
`ModelReferenceCompliant` string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports model reference.
`ParamNamingFcn`	Not available	For MathWorks use only.
`PostCodeGenCommand` *`string`* - `''`	Not available	Add the specified post code generation command to the model build process.
`PreserveName`	Not available	For MathWorks use only.
`PreserveNameWithParent`	Not available	For MathWorks use only.
`ProcessScript`	Not available	For MathWorks use only.
`ProcessScriptMode`	Not available	For MathWorks use only.
`SignalNamingFcn`	Not available	For MathWorks use only.
`SystemCodeInlineAuto`	Not available	For MathWorks use only.
`TargetFcnLib`	Not available	For MathWorks use only.
`TargetLibSuffix` *`string`* - `''`	Not available	Control the suffix used for naming a target's dependent libraries (for example, `_target.lib` or `_target.a`). If specified, the string must include a period (.). (For generated model reference libraries, the library suffix defaults to `_rtwlib.lib` on Windows systems and `_rtwlib.a` on UNIX systems.)

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`TargetPreCompLibLocation` *string* - ''	Not available	Control the location of precompiled libraries. If you do not set this parameter, the code generator uses the location specified in `rtwmakecfg.m`.
`TargetPreprocMaxBitsSint` int - **32**	Not available	Specify the maximum number of bits that the target C preprocessor can use for signed integer math.
`TargetPreprocMaxBitsUint` int - **32**	Not available	Specify the maximum number of bits that the target C preprocessor can use for unsigned integer math.

`TargetTypeEmulationWarn`
`SuppressLevel`
`SuppressLevel`
int - **0**
	Not available	When greater than or equal to 2, suppress warning messages that the Simulink Coder software displays when emulating integer sizes in rapid prototyping environments.

7

Model Advisor Checks

Embedded Coder Checks

In this section...

"Checks Overview" on page 7-3

"Check solver for code generation" on page 7-4

"Identify questionable blocks within the specified system" on page 7-6

"Identify lookup table blocks that generate expensive out-of-range checking code" on page 7-7

"Check output types of logic blocks" on page 7-9

"Identify blocks using one-based indexing" on page 7-10

"Check the hardware implementation" on page 7-11

"Identify questionable software environment specifications" on page 7-12

"Identify questionable code instrumentation (data I/O)" on page 7-14

"Check for blocks that have constraints on tunable parameters" on page 7-15

"Check for blocks not recommended for MISRA-C:2004 compliance" on page 7-17

"Check configuration parameters for MISRA-C:2004 compliance" on page 7-18

"Check for model reference configuration mismatch" on page 7-20

"Identify blocks that generate expensive saturation and rounding code" on page 7-21

"Check sample times and tasking mode" on page 7-22

"Identify questionable subsystem settings" on page 7-23

"Identify questionable fixed-point operations" on page 7-24

"Check model configuration settings against code generation objectives" on page 7-33

"Check for efficiency optimization parameters" on page 7-34

Checks Overview

Use Simulink Coder 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 Simulink Coder software from generating code. Underspecifying sample times can lead to undesired results. Avoid generating code that might corrupt data or produce unpredictable behavior.

Results and Recommended Actions

Condition	Recommended Action
The solver type is set incorrectly for model level code generation.	Set **Configuration Parameters** > **Solver** > • **Type** to `Fixed-step` • **Solver** to `Discrete (no continuous states)`
Multitasking diagnostic options are not set to `error`.	Set **Configuration Parameters** > **Diagnostics** > • **Sample Time** > **Multitask conditionally executed subsystem** to `error` • **Sample Time** > **Multitask rate transition** to `error` • **Data Validity** > **Multitask data store** to `error`

Tips

You do not have to modify the solver settings to generate code from a subsystem. The Embedded Coder build process automatically changes **Solver type** to `fixed-step` when you select **Code Generation** > **Build Subsystem** or **Code Generation** > **Generate S-Function** from the subsystem context menu.

See Also

- "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 Simulink Coder 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

"Supported Products and Block Usage"

Identify lookup table blocks that generate expensive out-of-range checking code

Identify lookup table blocks that generate code to protect against out-of-range inputs for breakpoint or index values.

Description

This check verifies that the following blocks do not generate code to protect against inputs that fall outside the range of valid breakpoint values:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

This check also verifies that all Interpolation Using Prelookup blocks do not generate code to protect against inputs that fall outside the range of valid index values.

Results and Recommended Actions

Condition	Recommended Action
The lookup table block generates out-of-range checking code.	Change the setting on the block dialog box so that out-of-range checking code is not generated. • For the 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table, and Prelookup blocks, select the check box for **Remove protection against out-of-range input in generated code**. • For the Interpolation Using Prelookup block, select the check box for **Remove protection**

Condition	Recommended Action
	against out-of-range index in generated code.

Action Results

Clicking **Modify** prevents lookup table blocks from generating out-of-range checking code, which makes the generated code more efficient.

See Also

- n-D Lookup Table block in the Simulink documentation
- Prelookup block in the Simulink documentation
- Interpolation Using Prelookup block in the Simulink documentation
- “Tips to Optimize Generated Code for Lookup Table Blocks” in the Simulink documentation

Check output types of logic blocks

Identify logic blocks that do not use `boolean` for the output data type.

Description

This check verifies that the output data type of the following blocks is `boolean`:

- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

Results and Recommended Actions

Condition	Recommended Action
The output data type of a logic block is not `boolean`.	In the block dialog box, set **Output data type** to `boolean`.

Action Results

Clicking **Modify** forces logic blocks to use `boolean` as the output data type. If a logic block uses `uint8` for the output type, clicking **Modify** changes the output type to `boolean`.

Identify blocks using one-based indexing

Identify blocks using one-based indexing.

Description

Zero-based indexing is more efficient in the generated code than one-based indexing. This check identifies blocks using one-based indexing.

See "cgsl_0101: Zero-based indexing".

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks configured for one-based indexing.	Configure the blocks for zero-based indexing. Update the supporting blocks.
The model or subsystem contains blocks requiring one-based indexing.	Evaluate the blocks to determine if one-based indexing is used. Consider replacing the blocks with Simulink basic blocks.

Check the hardware implementation

Identify inconsistent or underspecified hardware implementation settings

Description

The Simulink and Simulink Coder 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 inefficient or incorrect code generation for the target hardware.

Results and Recommended Actions

Condition	Recommended Action
Device type is set to `Unspecified (assume 32-bit Generic)`.	Set **Configuration Parameters > Hardware Implementation > Device type** to the target hardware.
Hardware implementation parameters are not set to recommended values.	Specify the following **Configuration Parameters** > **Hardware Implementation** parameters: • **Byte ordering** • **Signed integer division rounding**
Hardware implementation **Embedded hardware** settings do not match **Emulation hardware** settings.	Consider selecting the **Configuration Parameters** > **Hardware Implementation** > **None** check box, or modify the settings to match.

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** > **Code Generation** > **Symbols** > **Maximum identifier length** parameter to `31` characters.
Configuration Parameters > Code Generation > Interface parameters are not set to recommended values.	Clear the following parameters on the **Configuration Parameters** > **Code Generation** > **Interface** pane: • **Support: continuous time** • **Support: non-finite numbers** • **Support: non-inlined S-functions**
Configuration Parameters > Code Generation > Symbols parameters are not set to recommended values.	Set the **Configuration Parameters** > **Code Generation** > **Symbols** > **Generate scalar inlined parameters as** parameter to `Literals`.

Condition	Recommended Action
Support: variable-size signals is selected. This might lead to inefficient code.	If you do not intend to support variable-sized signals, clear the **Code Generation > Interface > "Support: variable-size signals" on page 6-172** check box in the Configuration Parameters dialog box.
The model contains Stateflow charts with weak Simulink I/O data type specifications.	Select the Stateflow chart property **Use Strong Data Typing with Simulink I/O**. You might need to adjust the data types in your model after selecting the property.

Limitations

A Stateflow license is required when using Stateflow charts.

See Also

"Strong Data Typing with Simulink I/O"

Identify questionable code instrumentation (data I/O)

Identify questionable code instrumentation.

Description

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

Results and Recommended Actions

Condition	Recommended Action
Interface parameters are not set to recommended values.	Set the **Configuration Parameters** > **Code Generation** > **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 **Code Generation** 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 **Code Generation** 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.	When tuning parameters during simulation or when running the generated code, you must: • Preserve monotonicity of the setting for the **Vector of input values** parameter. • Preserve the number and location of zero values that you specify for **Vector of input values** and **Vector of output values** parameters if you specify multiple zero values for the **Vector of input values** parameter.
Lookup Table (2-D) blocks have tunable parameters.	When tuning parameters during simulation or when running the generated code, you must: • Preserve monotonicity of the setting for the **Row index input values** and **Column index of input values** parameters. • Preserve the number and location of zero values that you specify for **Row index input values**, **Column index of input values**,

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

- 1-D Lookup Table
- 2-D Lookup Table

Check for blocks not recommended for MISRA-C:2004 compliance

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

Description

Following the recommendations of this check increases the likelihood of generating MISRA-C:2004 compliant code for embedded applications.

See “hisl_0020: Blocks not recommended for MISRA-C:2004 compliance”.

Results and Recommended Actions

Condition	Recommended Action
Blocks that are not supported or recommended for MISRA-C:2004 compliant code generation were found in the model or subsystem. For a list of blocks, see “hisl_0020: Blocks not recommended for MISRA-C:2004 compliance”.	Consider replacing the specified blocks.

Limitations

This check does not review libraries.

See Also

- “Developing Models and Code That Comply with MISRA C® Guidelines” in the Embedded Coder documentation.
- “MISRA-C:2004 Compliance Considerations” in the Simulink documentation.

Check configuration parameters for MISRA-C:2004 compliance

Identify configuration parameters that might impact MISRA-C:2004 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA-C:2004 compliant code for embedded applications.

See "hisl_0060: Configuration parameters that improve MISRA-C:2004 compliance".

Results and Recommended Actions

Condition	Recommended Action
Model Verification block enabling is set to `Use local settings` or `Enable All`.	In the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane, set **Model Verification block enabling** to `Disable All`.
System target file is set to a GRT-based target.	In the Configuration Parameters dialog box, on the **Code Generation > General** pane, set **System target file** to an ERT-based target.
Code Generation > Interface parameters are not set to the recommended values.	In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane: • Clear **Support: non-finite numbers** • Clear **Support: continuous time** (ERT-based target only) • Clear **Support: non-inlined S-functions** (ERT-based target only)

Condition	Recommended Action
	• Clear **MAT-file logging** • Set **Target function library** to `C89/C90 (ANSI)` (ERT-based target only)
Parenthesis level is not set to `Maximum (Specify precedence with parentheses)`.	In the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane, set **Parenthesis level** to `Maximum (Specify precedence with parentheses)`.
Maximum identifier length is not set to `31`.	In the Configuration Parameters dialog box, on the **Code Generation > Symbols** pane, set **Maximum identifier length** to `31`.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Limitations

This check does not review referenced models.

See Also

- "Developing Models and Code That Comply with MISRA C Guidelines" in the Embedded Coder documentation.
- "MISRA-C:2004 Compliance Considerations" in the Simulink documentation.

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

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 whether your application requires setting **Function Block Parameters** > **Signal Attributes** > **Saturate on integer overflow**. Otherwise, clear the **Saturate on integer overflow** parameter for 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 `SingleTasking`, or change the **Configuration Parameters** > **Hardware Implementation** settings.

See Also

"Single-Tasking and Multitasking Execution Modes"

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** > **Function packaging** option set to `Function`.	Set the **Subsystem Parameters** > **Function packaging** parameter to `Auto`.

See Also

Subsystem block

Identify questionable fixed-point operations

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

Description

The following operations can lead to nonoptimal results:

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

  - Blocks with relative scaling of inputs and outputs must determine the ideal product in the generated C code. The C code required to do this multiplication is large and slow.
  - Blocks that multiply signals with nonzero bias require extra steps to implement the multiplication. Inserting Data Type Conversion blocks remove the biases, and allow you to control data type and scaling for intermediate calculations. The conversion is done once and all blocks in the subsystem benefit from simpler, bias-free math.
  - Blocks are multiplying signals with mismatched 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 Simulink Coder 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 Simulink Coder 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.
  - 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.
  - When you select `isNan`, `isFinite`, or `isInf` as the operation for the Relational Operator block, the block switches to one-input mode. In this mode, if the input data type is fixed point, boolean, or a built-in integer, the output is always FALSE for `isInf` and `isNan`, TRUE for `isFinite`. This might result in dead code which will be eliminated by Simulink Coder.
- 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.
- The Compare to Zero block uses the input data type to represent zero. If the input data type of the Compare to Zero block cannot represent zero exactly, the input signal is compared to the closest representable value of zero, resulting in parameter overflow.
- The Compare to Constant block uses the input data type to represent its **Constant value** parameter. If the **Constant value** is outside the range that the input data type can represent, the input signal is compared to the closest representable value of the constant, resulting in parameter overflow.

Results and Recommended Actions

Conditions	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.

Conditions	Recommended Action
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 `fixpt_look1_func_approx`.
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 `fixpt_evenspace_cleanup`.
Lookup Table vector of input values is evenly spaced, but the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.
For a Prelookup or n-D Lookup Table block, **Index search method** is `Evenly spaced points`. Breakpoint data does not have power of 2 spacing.	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.
n-D Lookup Table breakpoint data is not evenly spaced and **Index search method** is not `Evenly spaced points`.	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`.
n-D Lookup Table breakpoint data is evenly spaced and **Index search method** is `Evenly spaced points`. But the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See `fixpt_look1_func_approx`.

Conditions	Recommended Action
n-D Lookup Table breakpoint data is evenly spaced, but the spacing is not a power of 2. Also, **Index search method** is not `Evenly spaced points`.	Set **Index search method** to `Evenly spaced points`. Also, if the data is nontunable, consider an even, power of 2 spacing.
n-D Lookup Table breakpoint data is evenly spaced, and the spacing is a power of 2. But the **Index search method** is not `Evenly spaced points`.	Set **Index search method** to `Evenly spaced points`.
Blocks require cumbersome multiplication.	Restrict multiplication operations: • So the product integer size is no larger than the target integer size. • To the recommended size.
Blocks multiply signals with nonzero bias.	Insert a Data Type Conversion block before and after the block containing the multiplication operation.
Product blocks are multiplying signals with mismatched slope adjustment factors.	Change the scaling of the output so that its slope adjustment factor is the product of the input slope adjustment factors.
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.	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.

Conditions	Recommended Action
	Note This optimization takes place only if certain simplicity and accuracy conditions are met. For more information, see "Handle Net Slope Correction" in the Simulink Fixed Point documentation.
Product blocks are configured to do multiple division operations.	Multiply all the denominator terms together, and then do a single division using cascading Product blocks.
Product blocks are configured to do many multiplication or division operations.	Split the operations across several blocks, with each block performing one multiplication or one division operation.
Product blocks are configured with a divide operation for the first input and a multiply operation for the second input.	Reverse the inputs so the multiply operation occurs first and the division operation occurs second.
An input with an invariant constant value is used as the denominator in an online division operation.	Change the operation to multiplication, and replace the invariant input by its reciprocal.
The data type range of the inputs of Sum blocks exceeds the data type range of the output, which can cause overflow or saturation.	Change the output and accumulator data types so the range equals or exceeds all input ranges. For example, if the model has two inputs • `int8` (–128 to 127) • `uint8` (0 to 255) The data type range of the output and accumulator must equal or

Conditions	Recommended Action
	exceed –128 to 255. A `int16` (–32768 to 32767) data type meets this condition.
A Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output.	Change the data types so the inputs, outputs, and accumulator have the same slope adjustment factor.
The net sum of the Sum block input biases does not equal the bias of the output.	Change the bias of the output scaling, making the net bias adjustment zero.
The inputs of the Relational Operator block have different data types.	• Change the data type and scaling of the invariant input to match other inputs. • Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type.
The inputs of the Relational Operator block have different slope adjustment factors.	Change the scaling of either input.
The output of the Relational Operator block is constant. This might result in dead code which will be eliminated by Simulink Coder.	Review your model design and either remove the Relational Operator block or replace it with the constant.
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.

Conditions	Recommended Action
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)`.
Parameter overflow occurred for the Compare to Zero block. This block uses the input data type to represent zero. The input data type cannot represent zero exactly, so the input value was compared to the closest representable value of zero.	Select an input data type that can represent zero.
Parameter overflow occurred for the following Compare to Constant block. This block uses the input data type to represent its **Constant value** parameter. The **Constant value** parameter is outside the range that the input data type can represent. The input signal was compared to the closest representable value of the **Constant value** parameter.	Choose an input data type that can represent the **Constant value** parameter or change the **Constant value** parameter to match the input data type.

Limitations

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

See Also

- 1-D Lookup Table
- n-D Lookup Table
- 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.

Results and Recommended Actions

Condition	Recommended Action
Parameters are set to values other than the value recommended for the specified objectives.	Set the parameters to the recommended values. **Note** A change to one parameter value can impact other parameters. Successfully passing the check might take multiple iterations.

Action Results

Clicking **Modify Parameters** changes the parameter values to the recommended values.

See Also

- The Simulink Coder "Recommended Settings Summary" on page 6-308
- The Embedded Coder "Recommended Settings Summary"
- "Application Objectives" in the Simulink Coder User's Guide.
- "Application Considerations" in the Embedded Coder documentation.

Check for efficiency optimization parameters

Identify optimization parameters that depend on the Execution efficiency or ROM efficiency objectives.

Description

Setting the optimization parameter **Use memcpy for vector assignment** to the recommended value increases the execution efficiency and reduces ROM usage.

Results and Recommended Actions

Condition	Recommended Action
The model specifies an execution or ROM efficiency objective and the **Use memcpy for vector assignment** parameter is cleared.	In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, select **Use memcpy for vector assignment**.

Action Results

Clicking **Modify** changes the parameter value to the recommended value.

Limitations

This check is in the Code Generation Advisor only.

See Also

- “Optimizing Code Generated for Vector Assignments”
- “Use memcpy for vector assignment” in the Simulink documentation

Index

A

activate 3-5
add 3-7
`addCompileFlags` function 3-2
`addDefines` function 3-10
`addIncludeFiles` function 3-13
`addIncludePaths` function 3-17
`addLinkFlags` function 3-20
`addLinkObjects` function 3-23
`addNonBuildFiles` function 3-28
address 3-31
`addSourceFiles` function 3-34
`addSourcePaths` function 3-38
`addTMFTokens` function 3-41
Async Interrupt block 5-2
Asynchronous Task Specification 5-10
Asynchronous Task Specification block 5-8

B

blocks
  Async Interrupt 5-2
  Generated S-Function 5-22
  Model Header
    reference 5-36
  Model Source
    reference 5-37
  Protected RT 5-38
  System Derivatives 5-39
  System Disable 5-40
  System Enable 5-42
    reference 5-41
  System Outputs 5-43
  System Start 5-44
  System Terminate 5-45
  System Update 5-46
  Task Sync 5-57
  Unprotected RT 5-69
Byte Pack block 5-14
Byte Reversal block 5-17
Byte Unpack block 5-19

C

compiler options
  adding to build information 3-2
  getting from build information 3-60
configuration parameters
  code generation 6-337
  Code Generation (general)
    Check model before generating code 6-33
  impacts of settings 6-308
  pane 6-276
    buildAction 6-279
    buildFormat 6-277
    Combine signal/state structures 6-208
    Compiler options string: 6-287
    DiagnosticActions 6-306
    Export IDE link handle to base
      workspace: 6-303
    Function name: 6-284
    Generate preprocessor
      conditionals 6-204
    Global types: 6-99
    gui item name 6-298
    IDE link handle name: 6-305
    ideObjBuildTimeout 6-300
    ideObjTimeout 6-302
    Linker options string: 6-289
    overrunNotificationMethod 6-282
    Profile real-time execution 6-294
    profileBy 6-296
    projectOptions 6-285
    System stack size (MAUs): 6-291
    variable-size signals 6-172
Configuration Parameters
  Code Generation (general)
    Check model 6-32
Configuration Parameters dialog box
  Code Generation (comments)

- Comments tab overview 6-63
- Custom comments 6-75
- Custom comments function 6-77
- Include comments 6-64
- MATLAB function help text 6-83
- MATLAB source code as comments 6-67
- Requirements in block comments 6-81
- Show eliminated blocks 6-69
- Simulink block descriptions 6-71
- Simulink block Stateflow object comments 6-66
- Simulink data object descriptions 6-73
- Stateflow object descriptions 6-79
- Verbose comments for Simulink global storage class 6-70

Code Generation (custom code)
- Custom Code tab overview 6-126
- Header file 6-131
- Include directories 6-134
- Initialize function 6-132
- Libraries 6-138
- Source file 6-130
- Source files 6-136
- Terminate function 6-133
- Use local custom code settings (do not inherit from main model) 6-128
- Use the same custom code settings as Simulation Target 6-127

Code Generation (debug)
- Debug tab overview 6-142
- Enable TLC assertion 6-149
- Profile TLC 6-145
- Retain .rtw file 6-144
- Start TLC coverage when generating code 6-148
- Start TLC debugger when generating code 6-146
- Verbose build 6-143

Code Generation (general)
- Build/Generate code 6-37
- Compiler optimization level 6-10
- Custom compiler optimization flags 6-12
- General tab overview 6-5
- Generate code only 6-35
- 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

Code Generation (interface)
- Block parameter access 6-194
- Block parameter visibility 6-190
- Configure C++ Encapsulation Interface 6-212
- Configure Model Functions 6-211
- Custom 6-158
- External I/O access 6-198
- Generate C API for parameters 6-222
- Generate C API for root-level I/O 6-224
- Generate C API for signals 6-221
- Generate C API for states 6-223
- Generate destructor 6-200
- Generate reusable code 6-183
- GRT compatible call interface 6-177
- interface 6-218
- Interface tab overview 6-154
- Internal data access 6-196
- Internal data visibility 6-192
- MAT-file logging 6-213
- MAT-file variable name modifier 6-216
- Maximum word length 6-175
- MEX-file arguments 6-227
- Multiword type definitions 6-173
- Pass root-level I/O as 6-188

Reusable code error diagnostic 6-186
Single output/update function 6-179
Static memory allocation 6-229
Static memory buffer size 6-231
Support absolute time 6-166
Support complex numbers 6-165
Support continuous time 6-168
Support floating-point numbers 6-161
Support non-finite numbers 6-163
Support non-inlined S-functions 6-170
Suppress error status in real-time model data structure 6-206
Target function library 6-155
Terminate function required 6-181
Transport layer 6-225
Use operator new for referenced model object registration 6-202
Utility code generation 6-159

Code Generation (report)
Code-to-model 6-46
Configure 6-50
Create code generation report 6-41
Eliminated / virtual blocks 6-51
Launch report automatically 6-44
Model-to-code 6-48
Report tab overview 6-40
Static Code Metrics 6-59
Traceable MATLAB functions 6-57
Traceable Simulink blocks 6-53
Traceable Stateflow objects 6-55

Code Generation (RSim target)
Enable RSim executable to load parameters from a MAT-file 6-236
Force storage classes to AUTO 6-238
RSim Target tab overview 6-235
Solver selection 6-237

Code Generation (S-function target)
Code Generation S-Function Target Tab Overview 6-241
Create new model 6-242
Include custom source code 6-244
Use value for tunable parameters 6-243

Code Generation (symbols)
Constant macros 6-105
#define naming 6-118
Field name of global types 6-94
Generate scalar inlined parameter as 6-111
Global types 6-91
Global variables 6-89
Local block output variables 6-103
Local temporary variables 6-101
M-function 6-114
Maximum identifier length 6-109
Minimum mangle length 6-107
Parameter naming 6-116
Reserved names 6-121
Signal naming 6-112
Subsystem methods 6-96
Symbols tab overview 6-88
Use the same reserved names as Simulation Target 6-120

Code Generation (Tornado target)
Base task priority 6-261
Code format 6-256
Download to VxWorks target 6-259
External mode 6-264
MAT-file logging 6-252
MAT-file variable name modifier 6-254
MEX-file arguments 6-268
Static memory allocation 6-270
Static memory buffer size 6-272
StethoScope 6-257
Target function library 6-248
Task stack size 6-263
Tornado Target tab overview 6-247
Transport layer 6-266
Utility code generation 6-250

Configuration Set Objectives dialog box 6-29

D

debug operation
new 3-87
debugging
and configuration parameter settings 6-308
derivatives
in custom code 5-39
disable code
in custom code 5-40

E

efficiency
and configuration parameter settings 6-308
enable code
in custom code 5-41
extensions, file. *See* file extensions

F

file and project operation
new 3-87
file extensions
updating in build information 3-137
file separator
changing in build information 3-140
file types. *See* file extensions
`findIncludeFiles` function 3-53

G

Generated S-Function block 5-22
`getCompileFlags` function 3-60
`getDefines` function 3-62
`getFullFileList` function 3-66
`getIncludeFiles` function 3-68
`getIncludePaths` function 3-71
`getLinkFlags` function 3-73
`getNonBuildFiles` function 3-76
`getSourceFiles` function 3-79
`getSourcePaths` function 3-82

H

header files
finding for inclusion in build information object 3-53

I

include files
adding to build information 3-13
finding for inclusion in build information object 3-53
getting from build information 3-68
include paths
adding to build information 3-17
getting from build information 3-71
initialization code
in custom code 5-42
interrupt service routines
creating 5-2

L

limitations
of Simulink Coder product 1-1
link objects
adding to build information 3-23
link options
adding to build information 3-20
getting from build information 3-73

M

macros
adding to build information 3-10
getting from build information 3-62
model header
in custom code 5-36
Model Header block

reference 5-36
Model Source block
reference 5-37
models
parameters for configuring 6-337

N

nonbuild files
adding to build information 3-28
getting from build information 3-76

O

outputs code
in custom code 5-43

P

`packNGo` function 3-91
parameter structure
getting 3-105
parameters
for configuring model code generation and targets 6-337
paths
updating in build information 3-137
program file, reload 3-100
project files
packaging for relocation 3-91
Protected RT block 5-38

R

rate transitions
protected 5-38
unprotected 5-69
reload 3-100
RSim target
parameter loading 6-236
`rsimgetrtp` function 3-105
`RTW.getBuildDir` function 3-116

S

S-function target
generating 5-22
safety precautions
and configuration parameter settings 6-308
separator, file
changing in build information 3-140
source code
in custom code 5-37
source files
adding to build information 3-34
getting from build information 3-79
source paths
adding to build information 3-38
getting from build information 3-82
startup code
in custom code 5-44
System Derivatives block 5-39
System Disable block 5-40
System Enable block 5-41
System Initialize block 5-42
System Outputs block 5-43
System Start block 5-44
System Terminate block 5-45
System Update block 5-46

T

Target Language Compiler and Function Library 3-134
Target Preferences block 5-47
targets
parameters for configuring 6-337
task function
creating 5-57
Task Sync block 5-57
termination code

in custom code 5-45
`tlc` function 3-134
TMF tokens
adding to build information 3-41
traceability
and configuration parameter settings 6-308

U

UDP Receive block 5-61
UDP Send block 5-66
Unprotected RT block 5-69
update code
in custom code 5-46
`updateFilePathsAndExtensions` function 3-137
`updateFileSeparator` function 3-140